Designing an Authentication System: a Dialogue in Four Scenes

Bill Bryant

Project Athena
Massachusetts Institute of Technology
Cambridge, MA 02139
billb@ATHENA.MIT.EDU

DRAFT, Februrary 8, 1988

ABSTRACT

This dialogue provides a fictitious account of the design of an open-network authentication system called "Charon." As the dialogue progresses, the characters Athena and Euripides discover the problems of security inherent in an open network environment. Each problem must be addressed in the design of Charon, and the design evolves accordingly. Athena and Euripides don't complete their work until the dialogue's close.

When they finish designing the system, Athena changes the system's name to "Kerberos," the name, coincidentally enough, of the authentication system that was designed and implemented at MIT's Project Athena. The dialogue's "Kerberos" system bears a striking resemblence to the system described in Kerberos: An Authentication Service for Open Network Systems presented at the Winter USENIX 1988, at Dallas, Texas.

Dramatis Personae:

Athena, an up and coming system developer.

Euripides, a seasoned developer and resident crank.

Scene I

A cubicle area. Athena and Euripides are working at neighboring terminals.

Athena: Hey Rip, this timesharing system is a drag. I can't get any work done because everyone else is logged in.

Euripides: Don't complain to me. I only work here.

Athena: You know what we need? We need to give everyone their own workstation so they don't have to worry about sharing computer cycles. And we'll use a network to connect all the workstations so folks can communicate with one another.

Euripides: Fine. So what do we need, about a thousand workstations?

Athena: More or less.

Euripides: Have you seen the size of a typical workstation's disk drive? There isn't enough room for all the software that you have on a timesharing machine.

Athena: I figured that out already. We can keep copies of the system software on various server machines. When you login to a workstation, the workstation accesses the system software by making a network connection with one of the servers. This setup lets a whole bunch of workstations use the same copy of the system software, and it makes software updates convenient. You don't have to trundle around to each workstation. Just modify the system software servers.

Euripides: All right. What are you going to do about personal files? With a timesharing system I can login and get to my files from any terminal that is connected to the system. Will I be able to walk up to any workstation and automatically get to my files? Or do I have to make like a PC user and keep my files on diskette? I hope not.

Athena: I think we can use other machines to provide personal file storage. You can login to any workstation and get to your files.

Euripides: What about printing? Does every workstation have its own printer? Whose money are you spending anyway? And what about electronic mail? How are you going to distribute mail to all these workstations?

Athena: Ah . . . Well obviously we don't have the cash to give everyone a printer, but we could have machines dedicated to print service. You send a job to a print server, and it prints it for you. You could do sort of the same thing with mail. Have a machine dedicated to mail service. You want your mail, you contact the mail server and pick up your mail.

Euripides: Your workstation system sounds really good Tina. When I get mine, you know what I'm going to do? I'm going to find out your username, and get my workstation to think that I am you. Then I'm going to contact the mail server and pick up your mail. I'm going to contact your file server and remove your files, and--

Athena: Can you do that?

Euripides: Sure! How are these network servers going to know that I'm not you?

Athena: Gee, I don't know. I guess I need to do some thinking.

Euripides: Sounds like it. Let me know when you figure it out.

Scene II

Euripides' office, the next morning. Euripides sits at his desk, reading his mail. Athena knocks on the door.

Athena: Well I've figured out how to secure an open network environment so that unscrupulous folks like you cannot use network services in other people's names.

Euripides: Is that so? Have a seat.

She does.

Athena: Before I describe it, can I lay down one ground rule about this discussion?

Euripides: What's your rule?

Athena: Well suppose I say something like the following: "I want my electronic mail, so I contact the mail server and ask it to send the mail to my workstation." In reality I'm not the entity that contacts the mail server. I'm using a program to contact the mail server and retrieve my mail, a program that is a CLIENT of the mail service program.

But I don't want to say "the client does such-and-such" every time I refer to a transaction between the user and a network server. I'd just as soon say "I do such-and-such," keeping in mind of course that a client program is doing things on my behalf. Is that okay with you?

Euripides: Sure. No problem.

Athena: Good. All right, I'll begin by stating the problem I have solved. In an open network environment, machines that provide services must be able to confirm the identities of people who request service. If I contact the mail server and ask for my mail, the service program must be able to verify that I am who I claim to be, right?

Euripides: Right.

Athena: You could solve the problem clumsily by requiring the mail server to ask for a password before I could use it. I prove who I am to the server by giving it my password.

Euripides: That's clumsy all right. In a system like that, every server has to know your password. If the network has one thousand users, each server has to know one thousand passwords. If you want to change your password, you have to contact all servers and notify them of the change. I take it your system isn't this stupid.

Athena: My system isn't stupid. It works like this: Not only do people have passwords, services have passwords too. Each user knows her or his password, each service program knows its password, and there's an AUTHENTICATION SERVICE that knows ALL passwords--each user's password, and each service's password. The authentication service stores the passwords in a single, centralized database.

Euripides: Do you have a name for this authentication service?

Athena: I haven't thought of one yet. Do you have any ideas?

Euripides: What's the name of that fellow who ferries the dead across the River Styx?

Athena: Charon?

Euripides: Yeah, that's him. He won't take you across the river unless you can prove your identity.

Athena: There you go Rip, trying to rewrite Greek mythology again. Charon doesn't care about your identity. He just wants to make sure that you're dead.

Euripides: Have you got a better name?

Pause.

Athena: No, not really.

Euripides: Then let's call the authentication service "Charon."

Athena: Okay. I guess I should describe the system, huh?

Let's say you want to use a service, the mail service. In my system you cannot use a service unless, ah, Charon tells the service that you are who you claim to be. And you can't get the okay to use a service unless you have authenticated yourself to Charon. When you request authentication from Charon, you have to tell Charon the service for which you want the okay. If you want to use the mail server, you've got to tell Charon.

Charon asks you to prove your identity. You do so by providing your secret password. Charon takes your password and compares it to the one that is registered for you in the Charon database. If the two passwords match, Charon considers your identity proven.

Charon now has to convince the mail server that you are who you say you are. Since Charon knows all service passwords, it knows the mail service's password. It's conceivable that Charon could give you the password, which you could forward to the mail service as proof that you have authenticated yourself to Charon.

The problem is, Charon cannot give you the password directly, because then you would know it. The next time you wanted mail, you could circumvent Charon and use the mail server without correctly identifying yourself. You could even pretend to be someone else, and use the mail server in that other person's name.

So instead of giving you the mail server's password, Charon gives you a mail service TICKET. This ticket contains a version of your username that has been ENCRYPTED USING the MAIL SERVER'S PASSWORD.

Ticket in hand, you can now ask the mail service for your mail. You make your request by telling the mail server who you are, and furnishing the ticket that proves you are who you say you are.

The server uses its password to decrypt the ticket, and if the ticket decrypts properly, the server ends up with the username that Charon placed in the ticket.

The service compares this name with the name you sent along with the ticket. If the names match, the mail server considers your identity proven and proceeds to give you your mail.

What do you think of those apples?

Euripides: I've got some questions.

Athena: I figured. Well go ahead.

Euripides: When a service program decrypts a ticket, how does it know that it has decrypted the ticket properly?

Athena: I don't know.

Euripides: Maybe you should include the service's name in the ticket. That way when a service decrypts a ticket, it can gauge its success on whether or not it can find its name in the decrypted ticket.

Athena: That sounds good to me. So the ticket looks something like this:

She scrawls the following on a pad of paper:

```
TICKET -  {username:servicename}
```

Euripides: So the service ticket contains just your username and the servicename?

Athena: Encrypted with the service's password.

Euripides: I don't think that's enough information to make the ticket secure.

Athena: What do you mean?

Euripides: Let's suppose you ask Charon for a mail server ticket. Charon prepares that ticket so that it has your username "tina" in it. Suppose I copy that ticket as it wizzes by on its way across the network from Charon to you. Suppose I convince my insecure workstation that my username is "tina." The mail client program on my workstation thinks I am you. In your name, the program forwards the stolen ticket to the mail server. The server decrypts the ticket and sees that it is valid. The username in the ticket matches the name of the user who sent the ticket. The mail server gives me your mail . . .

Athena: Oh! Well that's not so good.

Euripides: But I think I know a way to fix this problem. Or to at least provide a partial fix to it. I think Charon should include more information in the service tickets it produces. In addition to the username, the ticket should also include the NETWORK ADDRESS from which the user asked Charon for the ticket. That gives you an additional level of security.

I'll illustrate. Suppose I steal your mail ticket now. The ticket has your workstation's network address in it, and this address does not match my workstation's address. In your name I forward the purloined ticket to the mail server. The server program extracts the username and network address from the ticket and attempts to match that information against the username and network address of the entity that sent the ticket. The username matches, but the network address does not. The server rejects the ticket because obviously it was stolen.

Athena: Bravo, bravo! I wish I had thought of that.

Euripides: Well that's what I'm around for.

Athena: So the revised ticket design looks like this

She scrawls the following on a chalkboard:

```
TICKET -   {username:ws_address:servicename}
```

Athena: Now I'm really excited. Let's build a Charon system and see if it works!

Euripides: Not so fast. I have some other questions about your system.

Athena: All right. (Athena leans forward in her chair) Shoot.

Euripides: Sounds like I've got to get a new ticket every time I want to use a service. If I'm putting in a full day's work, I'll probably want to get my mail more than once. Do I have to get a new ticket every time I want to get my mail? If that's true, I don't like your system.

Athena: Ah . . . Well I don't see why tickets can't be reusable. If you get a ticket for the mail server, you ought to be able to use it again and again. For instance, when the mail client program makes a request for service in your name, it forwards a COPY of the ticket to the mail server.

Euripides: That's better. But I still have problems. You seem to imply that I have to give Charon my password every time I want to use a service for which I don't have a ticket. I login and want to access my files. I fire off a request to Charon for the proper ticket and this means that I've had to use my password. Then I want to read my mail. Another request to Charon, I have to enter my password again. Now suppose I want to send one of my mail messages to the print server. Another Charon request and, well you get the picture.

Athena: Uh, yeah, I do.

Euripides: And if that weren't bad enough, consider this: it sounds like when you authenticate yourself to Charon, you send your secret password over the network in cleartext. Clever people like yours truly can monitor the network and steal copies of people's passwords. If I've got your password, I can use any service in your name.

Athena sighs.

Athena: These are serious problems. Guess I need to go back to the drawing board.

Scene III

The next morning, Athena catches Euripides at the coffee area. She taps him on the shoulder as he fills his cup.

Athena: I've got a new version of Charon that solves our problems.

Euripides: Really? That was quick.

Athena: Well, you know, problems of this nature keep me up all night.

Euripides: Must be your guilty conscience. Shall we repair to yon small conference room?

Athena: Why not?

The two move to the small conference room.

Athena: I'll begin by stating the problems again, but I'll invert them so that they become requirements of the system.

Athena clears her throat.

Athena: The first requirement: Users only have to enter their passwords once, at the beginning of their workstation sessions. This requirement implies that you shouldn't have to enter your password every time you need a new service ticket. The second requirement: passwords should not be sent over the network in clear text.

Euripides: Okay.

Athena: I'll start with the first requirement: you should only have to use your password once. I've met this requirement by inventing a new network service. It's called the "ticket-granting" service, a service that issues Charon tickets to users who have already proven their identity to Charon. You can use this ticket-granting service if you have a ticket for it, a ticket-granting ticket.

The ticket-granting service is really just a version of Charon in as much as it has access to the Charon database. It's a part of Charon that lets you authenticate yourself with a ticket instead of a password.

Anyhow, the authentication system now works as follows: you login to a workstation and use a program called *kinit* to contact the Charon server. You prove your identity to Charon, and the *kinit* program gets you a ticket-granting ticket.

Now say you want to get your mail from the mail server. You don't have a mail server ticket yet, so you use the "ticket-granting" ticket to get the mail server ticket for you. You don't have to use your password to get the new ticket.

Euripides: Do I have to get a new "ticket-granting" ticket every time I need to get to another network service?.

Athena: No. Remember, we agreed last time that tickets can be reused. Once you have acquired a ticket-granting ticket, you don't need to get another. You use the ticket-granting ticket to get the other tickets you need.

Euripides: Okay, that makes sense. And since you can reuse tickets, once the ticket-granting service has given you a ticket for a particular service, you don't need to get that particular ticket again.

Athena: Yeah, isn't that elegant?

Euripides: Okay, I buy it so far . . . As long as you didn't have to send your password in cleartext over the network when you got the ticket-granting ticket.

Athena: Like I said, I've solved that problem as well. The thing is, when I say you have to contact Charon to get the ticket-granting ticket, I make it sound as though you have to send your password in cleartext over the network to the Charon Server. But it doesn't have to be that way.

Here's really what happens. When you use the *kinit* program to get the ticket-granting ticket, *kinit* doesn't send your password to the Charon server, *kinit* sends only your username.

Euripides: Fine.

Athena: Charon uses the username to look up your password. Next Charon builds a packet of data that contains the ticket-granting ticket. Before it sends you the packet, Charon uses your password to encrypt the packet's contents.

Your workstation receives the ticket packet. You enter your password. *Kinit* attempts to decrypt the ticket with the password you entered. If *kinit* succeeds, you have successfully authenticated yourself to Charon. You now possess a ticket-granting ticket, and that ticket can get you the other tickets you require.

How's that for some fancy thinking?

Euripides: I don't know . . . I'm trying to think myself. You know, I think the parts of the system that you just described work pretty well. Your system requires me to authenticate myself only once. Thereafter Charon can issue me service tickets without my being aware of it. Seamless, seamless in that regard. But there's something about the design of the service ticket that troubles me somehow. It has to do with the fact that tickets are reusable. Now I agree that they have to be reusable, but reusable tickets are, by their nature, very dangerous.

Athena: What do you mean?

Euripides: Look at it this way. Suppose you are using an insecure workstation. In the course of your login session you acquire a mail service ticket, a printing service ticket, and a file service ticket. Suppose you inadvertantly leave these tickets on the workstation when you logout.

Now suppose I login to the workstation and find those tickets. I'm feeling like causing trouble, so I make the workstation think that I am you. Since the tickets are made out in your name, I can use the mail client program to access your mail, I can use the file service client to access and remove your files, and I can use the printing command to run up huge bills on your account. All because these tickets have been accidentally left lying around.

And nothing can keep me from copying these tickets to a place of my own. I can continue to use them for all eternity.

Athena: But that's an easy fix. We just write a program that destroys a user's tickets after each login session. You can't use tickets that have been destroyed.

Euripides: Well obviously your system must have a ticket-destroying program, but it's foolish to make users rely on such a thing. You can't count on users to remember to destroy their tickets every time they finish a workstation session. And even if you rely upon your users to destroy their tickets, consider the following scenario.

I've got a program that watches the network and copies service tickets as they zip accross the network. Suppose I feel like victimizing you. I wait for you to begin a workstation session, I turn on my program and copy a bunch of your tickets.

I wait for you to finish your session, and eventually you logout and leave. I fiddle with my workstation's network software and change its address so that it matches the address of the workstation you were using when you acquired the tickets I copied. I make my workstation believe that I am you. I have your tickets, your username, and the correct network address. I can REPLAY these tickets and use services in your name.

It doesn't matter that you destroyed your tickets before you ended your workstation session. The tickets I have stolen are valid for as long as I care to use them, because your current ticket design does not place a limit on the number of times you can reuse a ticket, or on how long a ticket remains valid.

Athena: Oh I see what you're saying! Tickets can't be valid forever because they would then constitute a huge security risk. We have to restrict the length of time for which a ticket can be used, perhaps give each ticket some kind of expiration date.

Euripides: Exactly. I think each ticket needs to have two additional pieces of information: a lifespan that indicates the length of time for which the ticket is valid, and a timestamp that indicates the date and time at which Charon issued the ticket. So a ticket would look something like this:

Euripides goes to the chalkboard and scrawls the following:

```
TICKET   {username:address:servicename:lifespan:timestamp}
```

Euripides: Now when a service decrypts tickets, it checks the ticket's username and address against the name and address of the person sending the ticket, and it uses the timestamp and lifespan information to see if the ticket has expired.

Athena: All right. What kind of lifetime should the typical service ticket have?

Euripides: I don't know. Probably the length of a typical workstation session. Say eight hours.

Athena: So if I sit at my workstation for more than eight hours, all my tickets expire. That includes my ticket-granting ticket. So I have to reauthenticate myself to Charon after eight hours.

Euripides: That's not unreasonable is it?

Athena: I guess not. So we're settled--tickets expire after eight hours. Now I've got a question for you. Suppose I have copied YOUR tickets from the network--

Euripides: (*Eyes twinkling*) Aw, Tina! You wouldn't really do that would you?

Athena: This is just for the sake of argument. I've copied your tickets. Now I wait for you to logout. Suppose you have a doctor's appointment or a class to attend, so you end your workstation session after a couple of hours. You are a smart boots and have destroyed your copies of the tickets before logging out.

But I've stolen your tickets, and they are good for about six hours. That gives me ample time to pillage your files and print one thousand copies of whatever in your name.

See, the lifetime-timestamp business works fine in the event that a ticket thief chooses to replay the ticket after the ticket has expired. If the thief can replay the ticket before that . . .

Euripides: Uh, well . . . Of course you are right.

Athena: I think we have run into a major problem. *She sighs.*

Pause.

Euripides: I guess that means you'll be busy tonight. Want more coffee?

Athena: Why not.

The two head for the coffee machine.

Scene IV

The next morning in Euripides' office. Athena knocks on the door.

Euripides: You've got rings under your eyes this morning.

Athena: Well, you know. Another one of those long nights.

Euripides: Have you solved the replay problem?

Athena: I think so.

Euripides: Have a seat.

She does.

Athena: As usual, I feel compelled to restate the problem. Tickets are reusable within a limited timespan, say eight hours. If someone steals your tickets and chooses to replay them before they expire, we can't do anything to stop them.

Euripides: That's the problem.

Athena: We could beat the problem if we designed the tickets so they couldn't be reusable.

Euripides: But then you would have to get a new ticket every time you wanted to use a network service.

Athena: Right. That is a clumsy solution at best. (Pause.) Ah, how do I proceed with my argument? (She ponders for a moment.)

All right, I'm going to restate the problem again, this time in the form of a requirement. A network service must be able to prove that the person using a ticket is the same person to whom that ticket was issued.

Let me trace the authentication process again and see if I can tease out an appropriate way to illustrate my solution to this problem.

I want to use a certain network service. I access that service by starting a client program on my workstation. The client sends three things to the service machine-- my name, my workstation's network address, and the appropriate service ticket.

The ticket contains the name of the person it was issued to and the address of the workstation that person was using when he or she acquired the ticket. It also contains an expiration date in the form of a lifespan and a timestamp. All this information has been encrypted in the service's Charon password.

Our current authentication scheme relies on the following tests:

- Can the service decrypt the ticket?
- Has the ticket expired?
- Do the name and workstation address specified in the ticket match the name and address of the person who sent the ticket?

What do these tests prove? The first test proves that the ticket either did or did not come from Charon. If the ticket cannot be decrypted, it did not come from the real Charon. The real Charon would have encrypted the ticket with the service's password. Charon and the service are the only two entities that know the service's password. If the ticket decrypts successfully, the service knows that it came from the real Charon. This test prevents folks from building fake Charon tickets.

The second test checks the ticket's lifespan and timestamp. If it has expired, the service rejects the ticket. This test stops people from using old tickets, tickets that perhaps were stolen.

The third test checks the ticket-user's name and address against the name and address of the person specified in the ticket. If the test fails, the ticket-user has obtained (perhaps surreptitiously) another person's ticket. The ticket is of course rejected.

If the names and addresses do match, what has the test proved? Nothing. Scallywags can steal tickets from the network, change their workstation addresses and usernames appropriately, and rifle other folks resources. As I pointed out yesterday, tickets can be replayed as long as they haven't expired. They can be replayed because a service cannot determine that the person sending the ticket is actually the ticket's legitimate owner.

The service cannot make this determination because it does not share a secret with the user. Look at it this way. If I'm on watch at Elsinore, you know, the castle in *Hamlet*, and you are supposed to relieve me, I'm not supposed to let you take my place unless you can provide the correct password. That's the case where the two of us share a secret. And it's probably a secret that someone else made up for everyone who stands on watch.

So I was thinking last night, why not have Charon make up a password for the legitimate ticket-owner to share with the service? Charon gives a copy of this SESSION KEY to the service, and a copy to the user. When the service receives a ticket from a user, it can use the session key to test the user's identity.

Euripides: Wait a second. How is Charon going to give both parties the session key?

Athena: The ticket-owner gets the session key as part of the reply from Charon. Like this:

She scrawls the following on a chalkboard:

```
CHARON REPLY  -  [sessionkey|ticket]
```

The service's copy of the session key comes inside the ticket, and the service gets the key when it decrypts the ticket. So the ticket looks like this:

```
TICKET -  {sessionkey:username:address:servicename:lifespan:timestamp}
```

When you want to get to a service, the client program you start builds what I call an AUTHENTICATOR. The authenticator contains your name and your workstation's address. The client encrypts this information with the session key, the copy of the session key you received when you requestesed the ticket.

```
AUTHENTICATOR - {username:address} encrypted with session key
```

After building the authenticator, the client sends it and the ticket to the service. The ticket cannot decrypt the authenticator yet because it doesn't have the session key. That key is in the ticket, so the service first decrypts the ticket.

After decrypting the ticket, the service ends up with the the following information:

- The ticket's lifespan and timestamp;
- The ticket-owner's name;
- The ticket-owner's network address;
- The session key.

The service checks to see if the ticket has expired. If all is well in that regard, the service next uses the session key to decrypt the authenticator. If the decryption proceeds without a hitch, the service ends up with a username and a network address. The service tests this information against the name and address found in the ticket, AND the name and address of the person who sent the ticket and authenticator. If everything matches, the service has determined that the ticket-sender is indeed the ticket's real owner.

Athena pauses, clears her throat, drinks some coffee.

I think the session key-authenticator business takes care of the replay problem.

Euripides: Maybe. But I wonder . . . To break this version of the system, I must have the proper authenticator for the service.

Athena: No. You must have the authenticator AND the ticket for the service. The authenticator is worthless without the ticket because the service cannot decrypt the authenticator without first having the appropriate session key, and the service cannnot get the appropriate session key without first decrypting the ticket.

Euripides: Okay, I understand that, but didn't you say that when a client program contacts the server, it sends the ticket and matching authenticator together?

Athena: Yes, I guess I said that.

Euripides: If that's what actually happens, what prevents me from stealing the ticket and authenticator at the same time? I'm sure I could write a program to do the job. If I've got the ticket and its authenticator, I believe I can use the two as long as the ticket has not expired. I just have to change my workstation address and username appropriately. True?

Athena: (*Biting her lip*) True. How dispiriting.

Euripides: Wait, wait, wait! This isn't such a big deal. Tickets are reusable as long as they haven't expired, but that doesn't mean that authenticators have to be reusable. Suppose we design the system so that authenticators can only be used once. Does that buy us anything?

Athena: Well, it might. Let's see, the client program builds the authenticator, then sends it with the ticket to the service. You copy both ticket and authenticator as they move from my workstation to the server. But the ticket and authenticator arrive at the server before you can send your copies. If the authenticator can only be used once, your copy of it is no good, and you lose when you attempt to replay your ticket and authenticator.

Well, that's a relief. So all we have to do is invent a way to make the authenticator a one-time usable thing.

Euripides: No problem. Let's just put a lifespan and timestamp on them. Suppose each authenticator has a lifespan of a couple of minutes. When you want to use a service, your client program builds the authenticator, stamps it with the current time, then sends it and the ticket to the server.

The server receives the ticket and authenticator and goes about its business. When the server decrypts the authenticator, it checks the authenticator's lifespan and timestamp. If the authenticator hasn't expired, and everything else checks properly, the server considers you authenticated.

Suppose I copied the athenticator and ticket as they crossed the network. I have to change my workstation's network address and my username, and I have to do this all in a couple of minutes. That's a pretty tall order. In fact I don't think it's possible. Unless . . .

Well, here's a potential problem. Suppose that instead of copying the ticket and authenticator as they travel from your workstation to the server, I copy original ticket packet that comes from Charon, the packet you receive when you ask Charon to give you a ticket.

This packet, as I recall, has two copies of the session key in it: one for you and one for the service. The one for the service is hidden in the ticket and I can't get to it, but what about the other one, the one you use to build authenticators?

If I can get that copy of the session key, I can build my own authenticators, and if I can build my own authenticators, I can break the system.

Athena: That's something I thought about last night, but then I traced the process of acquiring tickets and found that it wasn't possible to steal authenticators that way.

You sit down at a workstation and use the *kinit* program to get your ticket-granting ticket. *Kinit* asks for your username, and after you enter it, *kinit* forwards the name to Charon.

Charon uses your name to look up your password, then proceeds to build a ticket-granting ticket for you. As part of this process, Charon creates a session key that you will share with the ticket-granting service. Charon puts a copy of the session key in the ticket-granting ticket, and puts your copy in the the ticket packet that you are about to receive. But before it sends you this packet, Charon encrypts the whole thing with your password.

Charon sends the packet across the network. Someone can copy the packet as it goes by, but they can't do anything with it because it has been encrypted with your password. Specifically, no one can steal the ticket-granting session key.

Kinit receives the ticket packet and prompts you for a password, which you enter. If you enter the correct password, *kinit* can decrypt the packet and give you your copy of the session key.

Now that you've taken care of the *kinit* business, you want to get your mail. You start the mail client program. This program looks for a mail service ticket and doesn't find one (after all, you haven't tried to get your mail yet). The client must use the ticket-granting ticket to ask the ticket-granting service for a mail service ticket.

The client builds an authenticator for the ticket-granting transaction and encrypts the authenticator with your copy of the ticket-granting session key. The client then sends Charon the authenticator, the ticket-granting ticket, your name, your workstation's address, and the name of the mail service.

The ticket-granting service receives this stuff and runs through the authentication checks. If everything checks properly, the ticket-granting service ends up with a copy of the session key that it shares with you. Now the ticket-granting service builds you a mail service ticket, and during this process, creates a new session key for you to share with the mail service.

The ticket-granting service now prepares a ticket packet to send back to your workstation. The packet contains the ticket and your copy of the mail service session key. But before it sends the packet, the ticket-granting service encrypts the packet with its copy of the TICKET-GRANTING session key. That done, the packet is sent on its way.

So here comes the mail service ticket packet, loping across the network. Suppose some network ogre copies it as it goes by. The ogre is out of luck because the packet is encrypted with the ticket-granting session key; you and the ticket-granting service are the only entities that know this key. Since the ogre cannot decrypt the mail ticket packet, the ogre cannot discover the MAIL SESSION KEY. Without this session key, the ogre cannot use any of the mail service tickets you might subsequently send across the network.

So I think we're safe. What do you think?

Euripides: Perhaps.

Athena: Perhaps! Is that all you can say!

Euripides: (*laughing*) Don't get upset. You should know my ways by now. I guess it is mean of me, and you up half the night.

Athena: Pthhhhh!

Euripides: All right, three-quarters of the night. Actually, the system is beginning to sound acceptible. This session key business solves a problem that I thought of last night: the problem of mutual authentication.

Pause.

Mind if I talk for a minute?

Athena: (*A trifle coldly*) Be my guest.

Euripides: You are so kind. *Euripides clears his throat.* Last night, while visions of session keys and authenticators danced in your head, I was trying to find new problems with the system, and I found one that I thought was pretty serious. I'll illustrate it by way of the following scenario.

Suppose you are sick of your current job and have determined that it is in your best interest to move on. You want to print your resume on the company's wizz-bang laser printer so that headhunters and potential employers can take note of your classiness.

So you enter the printing command, and direct it to send the resume to the appropriate print server. The command gets the proper service ticket, if you don't already have it, then sends the ticket in your name to the appropriate print server. At least that's where you think it's headed. You don't in fact know that the request is headed for the right print server.

Suppose that some unscrupulous hacker--say it's your boss--has screwed system around so that he redirects your request and its ticket to the print server in his office. His print service program doesn't care about the ticket or its contents. It throws away the ticket and sends a message to your workstation indicating that the ticket passed muster, and that the server is ready and willing to print your job. The printing command sends the job to the fraudulant print server and the enemy ends up with your resume.

I'll state the problem by way of contrast. Without session keys and authenticators, Charon can protect its servers from false users, but it cannot protect its users from false servers. The system needs a way for client programs to authenticate the server before sending sensitive information to the service. The system must allow for MUTUAL AUTHENTICATION.

But the session key solves this problem as long as you design your client programs properly. Back to the print server scenario. I want a print client program that makes sure the service it's sending jobs to is the legitimate service.

Here's what such a program does. I enter the printing command and give it a filename, the name of my resume. Assume that I have a print service ticket and session key. The client program uses the session key to build an authenticator, then sends the authenticator and ticket to the "supposed" print server. The client DOES NOT send the resume yet; it waits for a response from the service.

The real service receives the ticket and authenticator, decrypts the ticket and extracts the session key, then uses the session key to decrypt the authenticator. This done, the service runs all the appropriate authentication tests.

Assume the tests confirm my identity. Now the server prepares a reply packet so that it can prove its identity to the client program. It uses its copy of the session key to encrypt the reply packet, then sends the packet to the waiting client.

The client receives the packet and attempts to decrypt it with my copy of the session key. If the packet decrypts properly and yields the correct server response message, my client program knows that the server that encrypted the packet is the real server. Now the client sends the resume job to the print service.

Suppose my boss screwed around the system so that his print server poses as the one I want. My client sends the authenticator and ticket to the "print service" and waits for a response. The fake print service cannot generate the correct response because it cannot decrypt the ticket and get the session key. My client will not send the job unless it receives the correct response. Eventually the client gives up waiting and exits. My print job does not get completed, but at least my resume did not end up on the desk of the enemy.

You know, I think we have a solid basis on which to implement the Charon Authentication System.

Athena: Perhaps. Anyway, I don't like the name "Charon."

Euripides: You don't? Since when?

Athena: I've never liked it, because the name doesn't make sense. I was talking to my Uncle Hades about it the other day, and he suggested another name, the name of his three-headed watch dog.

Euripides: Oh, you mean "Cerberus."

Athena: Bite your tongue Rip! "Cerberus" indeed . . .

Euripides: Er, isn't that the name?

Athena: Yeah, if you happen to be a Roman! I'm a Greek goddess, he's a Greek watch dog, and his name is "Kerberos," "Kerberos" with a K.

Euripides: Okay, okay, don't throw thunderbolts. I'll buy the name. Actually, it has a nice ring to it. Adios Charon and hello to Kerberos.