PROFESSOR: So today, we're going to talk about Kerberos, which is a cryptographically secure, in some ways, protocol for authenticating computers and applications to one another over the network. So this is now finally going to use cryptography whereas last lecture, we looked at just using these TCP sequence numbers to provide security. So before we dive into the details, I want to mention there's two bits of administrivia that you might want to know.

There's a quiz review today 7 to 9 o'clock in some room, I think 32 or 23. It's on schedule page and the quiz is next Wednesday. And also, we want you guys to post some idea for your final project on Piazza by tomorrow just so that your classmates can know what you're thinking and help you form groups. All right, so let's talk about Kerberos, all right?

So what is the setting that Kerberos is trying to support, here? So the model for Kerberos, what these guys had in mind when they were designing Athena guess 25 or 30 years ago now, is that they were imagining they were going to have a network with many server machines and many client machines interacting with one another. So you could imagine, well, you have a file server somewhere out there. You probably have a mail server connected to the network, maybe other services-- printer.

And all these are just connected to some sort of a network rather than being processes on a single machine. So the prelude to Athena and Kerberos is that you just have a time sharing machine where everything was a single process and everyone would just log into the same system and store their files there. So these guys wanted a more distributed system.

So this meant that you'd have these servers on one side and you'd also have a bunch of workstations that users would use themselves which would run applications and potentially connect to these servers and store the user's files, get their mail et cetera. And the problem that they wanted to solve was how to authenticate the users that are using these workstations to all these different servers in the back end without having to trust the network to be correct, which seems like a sensible design requirement in many ways. And I should mention that I guess the alternative to Kerberos at the time was these R login commands that we're looking at in last lecture, which seems like a bad plan. They just use IP addresses to authenticate the users.

And Kerberos was reasonably successful. It actually still is used at MIT. And actually, it's the basis of Microsoft's Active Directory server. So pretty much every Microsoft based sort of Windows Server uses Kerberos in some form or another.

But of course, because this protocol was designed 25 or 30 years ago, requirements change. What people assume change. People now understand much more about security.

So their version of Kerberos that's in use today is noticeably different in many ways from the version in the paper. And we'll look at exactly what assumptions aren't good enough anymore today and what did they get wrong. It's sort of inevitable for any protocol that was the first protocol to really use cryptography to authenticate parties over the network in this sort of full system scale.

Anyway, so that's the sort of setting for Kerberos. And it might be interesting to figure out what's the trust model, right? So Kerberos is going to introduce this extra Kerberos server sitting on the side.

So our third model at some level is that the network is untrusted like we were thinking about in last lecture. But who do we have to trust in this Kerberos setting? So of course, one thing is everyone-- all parties have to trust the Kerberos server.

So that's an assumption these guys were willing to make at the time that this

Kerberos server would be in charge of all network authentication in some form or another. Do we have anyone else that has to trust anything in the setting? For example, yeah.

STUDENT: Well users should trust their own machines.

PROFESSOR: Yes, so that's a good point, right? There's the users that I didn't draw here. But these guys are using some workstation.

And it's actually pretty important in Kerberos that the user trusts their workstation. So what goes wrong if you don't trust your workstation? Or presumably, it can just sniff your password and do whatever you type-- you know, LS, it tries-- runs RMX. That seems kind of unfortunate. Yeah.

STUDENT: But it's more than that though, because they could even, like, sniff your Kerberos ticket.

PROFESSOR: Yeah, exactly. Or when you log in, you type in your password, which is even worse than the ticket. Yeah, absolutely, yeah.

So this is actually a bit of a problem with Kerberos in the sense that if you don't trust the workstation, then you're in a bit of trouble. So if you have your own laptop, this seems like a sensible assumption to make. If you're using a public computer, this is a bit more questionable. And we'll see exactly what could go wrong. Yeah.

STUDENT: You have to trust the people administrating aren't doing anything bad with [INAUDIBLE] servers and giving them privilege access to one another.

STUDENT: So what do you mean? So of course, like, ISNT is going to run these machines down here. But I think the machines themselves don't necessarily have to trust one another. So the mail server doesn't trust the print server or the file server necessarily.

STUDENT: Not to trust, but he might be able to access a server you're not supposed to have access to by going through another server.

PROFESSOR: Yeah, that's true, I guess. Yeah, so if you set up any trust relationship between these guys-- like, if you give the mail server some back end to go access your files just for convenience, then this could be abused, yeah. So you have to be careful about not introducing additional sort of levels of trust or trust relations here.

All right, anything else that matters here? Do the servers have to trust the users in any way or the workstations? No, I guess presumably not. So this was the whole goal that the server doesn't have to a priori even know necessarily what all these users or how to authenticate them or what this workstation is doing until it can cryptographically prove that this is a legitimate user and they should have access to their data or whatnot.

All right, so let's look at how does Kerberos work or what's the overall architecture, at least. So the plan that these guys had in mind is that there would be this Kerberos server. We have drawn it up there but let's draw it in a slightly bigger scale. So this is the Kerberos server.

And today, it's typically called KDC-- Key Distribution Center. And there's all these users out here somewhere and also services that you might want to connect to. And the plan is that the Kerberos server is going to be responsible for storing a shared key between the Kerberos server and every entity in the world, or in this realm at least. So if the user has some sort of a key KC for client, than the Kerberos server is going to remember this key somewhere here.

And similarly for the server, the key KS is going to be known to the service itself and to the Kerberos server but hopefully no one else. So you can think of it as like a generalization of passwords, right? So you know a password and the Kerberos server knows your password but no one else.

And this is how you guys are going to prove to each other, yeah, I'm the right guy. I know this password and no one else does. Makes sense?

And the other thing the Kerberos server is going to have to do is, of course, keep track of who is it that owns this key, right? So it's going to have this table mapping

some sort of a name. So this is some sort of a user. This is maybe service maybe AFS or something like this.

And the KDC is responsible for storing the gigantic table-- well, not very large in terms of the number of bytes, necessarily, but one entry per entity at MIT that the Kerberos server knows about. Makes sense? All right, and then we're going to have sort of provide two interfaces, right?

The paper is a little fuzzy above this or it pretends like there's really two services. But really, what's going on is that there's two interfaces to the same machine. One of them is called Kerberos in the paper and one of them is called TGS for Ticket Granting Service.

And really, these are just two ways of talking to the same thing in the back end. And the protocol is a little different for these things. So initially, when the user logs in, they're going to talk to this guy over here. And they're going to send their client name, C. So this might be your Athena username.

And the server is going to respond to back with a ticket, a TGS or-- well, some sort of a ticket. We'll look at the details in a bit. And then when you want to talk to some server down here, KS, then you're going to talk to this TGS first and say, oh, hey, I already logged in through the Kerberos interface. Now, I want to talk to the server S.

So you'd tell the TGS about the server you want to talk to. And then it returns you back some sort of a ticket for talking to the server S. And then you can finally start talking to the server over here by initially passing it this ticket for S.

Does this all make sense? This is the sort of high level plan. So why do these guys have two interfaces?

Well, I guess I wanted to actually ask a bunch of questions. Like, in the case of a service, this service is probably going to be stored on disk. What's going on with this KC on the user side? Where does KC come from in Kerberos? Yeah.

STUDENT: KDMS [INAUDIBLE] the database.

PROFESSOR: Yeah, well, the key C sits here. And that's absolutely true. It's sits on this giant database. But it also has to be known to the user because the user has to prove that they are the user. Yeah.

STUDENT: Is that a one way function and then the password?

PROFESSOR: Yeah, so they actually have this sort of cute plan where the KC is actually going to be derived by hashing the user's password or really some sort of key duration function. And there's several different ones of perverse uses. But you're basically going to take a password, transform it in some way, and get this key KC. All right, so that seems good.

Why do we need two protocols, right? You could imagine that you just always ask the Kerberos server for tickets directly this way. You say, well, hey, I want a ticket for this particular principle name. And it'll send you back a ticket and you can decrypt it with your KC afterwards. Yeah.

STUDENT: Can you [INAUDIBLE] or ask the user to reenter their password each time they want to [INAUDIBLE] service?

PROFESSOR: Right, so the reason for the difference between these two interfaces is that on this interface, all the responses come back encrypted with your key KC. And the Kerberos designers are a little worried about keeping this KC around for a long time. Because either you have to ask the user to enter it, which is just annoying for the user, or it sits around in memory.

And this is basically as good as the user's password. So if this gets disclosed, then someone with access to KC can keep accessing the user's files until the user maybe changes their password and potentially even longer. We'll see about that.

So this KC is a really dangerous thing to leak. So the whole point of using this interface first and using this interface later for all subsequent requests is that you can actually forget KC as soon as you decrypt this TGS response from the Kerberos interface. And from that point on, even if you leak it, there is a lifetime associated

with this ticket. So worst case, someone gets access to your account for a couple of hours, not for an unbounded amount of time. So that's the sort of main difference why you guys have this slightly more complicated picture with two ways of accessing the same thing.

All right, so before we dive into the mechanics of how these protocols actually look like on the wire, let's talk a little bit about this naming aspect in Kerberos, right? So at some level, you could think of Kerberos as being a name registry, right? So it's really responsible for mapping these cryptographic keys onto string names.

And this is the fundamental sort of operation that Kerberos needs to provide. In fact, you'll see in the next lecture even on the web, we need some function like this. It's implemented differently from Kerberos but this is a fundamentally very important thing to have in almost any distributed system for security. So let's look at how Kerberos actually deals with names.

So in Kerberos, the sort of system calls every entity in this database a principal And a principal in Kerberos is actually just a string, right? So you can actually have some principal like, I don't know, nickolei.

So that's a string. And that could be a principal in some Kerberos realm. So it would literally be the thing that sits in this left column of the KDC's table.

And there's also some extra instances that the protocl supports. I could say, you know, nikolai.extra secure or something. And I might use this as a different entity for machines I really care about. So maybe I will have a different password for really secure things and a different password for my regular account. So this is just sort of how Kerberos-- this is what the paper talks about with instances.

So one might actually wonder-- where do you actually see instances? Where do influences come from? So the Kerberos service maps names to keys for you, but how do you know which name to ask for or which name to expect when you are talking to some machine?

So I guess what I'm asking is, what are names appear outside of the Kerberos

machine. So I guess we could ask, OK, where do user names appear? Any ideas? Yeah.

STUDENT: You can ask the MIT server for usernames presumably.

PROFESSOR: Right, yeah. So you could enumerate these things. Also, the users just type them in when they log into a machine.

So that's where it initially comes from. Do usernames appear anywhere else? Should they appear anywhere else? Yeah.

STUDENT: Possibly access the [INAUDIBLE] lists on the various services.

PROFESSOR: Yes, that's actually an important point, right? The goal of Kerberos is just to map keys to names. But it doesn't tell you what that name should have access to.

In fact, the way the applications typically use Kerberos is that one of these servers uses Kerberos to figure out, OK, what string name am I talking to? So when the mail server gets a connection from some workstation and it get the Kerberos ticket that prove that maybe this user is called Nikolai, then the mail server internally now has to figure out, OK, well, what should that guy have access to? And same for a file server.

So inside of all these servers, there's probably things like access control lists, maybe groups, maybe other things that actually do the authorization step. So Kerberos provides authentication which tells you who is this person I'm talking to. And the service itself is responsible for implementing the authorization part where they decide what access you should have based on your username here.

Makes sense? All right, so that's where the user names appear. There's also other principal names the Kerberos supports for services, right?

So services, I guess the paper suggests, look something like this. That's rcmd.hostname. And the reason that you need a name for one of these services is that you want to know, for example, when I connect to a file server, I actually want

mutual authentication.

It's not just the final server learns who I am, but I, the user or the workstation, want to be convinced that I'm talking to the right file server and not some fake file server that's impersonating my files. Because maybe I'll look at the grades file and submit it to the registrar. It would be too bad if some file server can impersonate the response and give me the wrong grades file all of a sudden.

So this is why the service principles also need their own name and the workstations need to figure out what name should I expect to see when I connect to the service. And typically, this comes from the user at some level. So for example, if I type SSH some machine foo, then this means that I should be expecting a Kerberos principal called rcmd.foo on the other end of this connection. And if it turns out to be someone else, this SSH client should abort and not let me connect because then I will be misled into talking to some other machine. That make sense?

So here's one interesting question. When can we reuse names in Kerberos? It's like, all of you guys have Athena accounts.

And when you graduate, could MIT, like, wipe out your database entry and allow someone else to register that same username? Would that be a good idea? Well, aside from the fact that you guys want accounts. Yeah.

STUDENT: Updated services as well so that they would, like map that username to [INAUDIBLE] permission theoretically?

PROFESSOR: Yeah, because these guys are actually just string entries somewhere in some ACL on a file server on a mail server. And just because you wipe out this entry in the Kerberos database doesn't mean that this entry is gone. And they're not versioning in any way, right?

This entry could literally say, you know, Alice has access to some Athena locker. And if this Alice graduates and her entry gets removed, then some new Alice comes along, registers, in the Kerberos database. But she gets a principal that looks identical to the old Alice. It's the same string.

So all of a sudden, the file server will give access to the new Alice to old Alice's data. So there's a bit of a complicated process for reclaiming principal names in Kerberos because there's no real connection or versioning between these guys. So as a result, it's actually kind of hard to reuse principal names. Once you register a principal, you probably don't want to reuse it very often.

And same for, in some sense, these principal names for service as well. As long as this hostname remain some well-known service that people expect to function in a certain way, you probably don't want to get rid of its key even if the service goes on. Because maybe a year later, some guy tries to connect to it and expects certain things.

And if it's been reused for a different service, that guy can impersonate things. Probably not as dramatic or as bad, but still, you have to be careful with reusing principal names in this kind of protocol. Makes sense?

Any questions? All right, so let's look at how the protocol itself now works. So we'll look first and this step of the protocol where you initially get your ticket with your password. And then we'll look at how this TGS interface works and how it differs then a little bit.

All right, so I guess there's this main data structure that Kerberos uses called a ticket. And this ticket looks like this. So there's a ticket to between a client and a server.

And this guy is basically the names of the server and the client that we're talking about-- the IP address of the client, some kind of timestamp, and an expiration time for how long the stick is valid. And there's also a key, KCS, that's going to be shared between the client and the server. So that's what's in a ticket.

And there's also this other weird data structure that Kerberos introduces called an authenticator. And an authenticator goes with a particular client C. And this thing is just the client's name, the IP address of the client, and the time stamp when the client generated this authenticator. And typically, both of these things are encrypted.

And authenticator is typically scripted with the key between the client on the server. So the authenticator sort of has to do with a particular connection between a client and a server. And the Kerberos ticket itself here is typically encrypted with the key for the service KS. So the subscript notation denotes here encryption with a particular key.

All right, so what does this-- so using this sort of notation here, let's try to figure out what is the protocol. By which the user initially logs into this Kerberos and gets their TGS ticket. So as we saw here before, right, the plan is the client is going to send their username over to the Kerberos server or that interface. And the response is going to be a ticket.

And what precisely the client actually sends over-- both the username C of the client that's issuing the request as well as the principal name-- well, the client is also a principal name. But the client also sends the principal name of the service for which it would like to get a ticket. And typically, the service name is actually the service name of this TGS guy over here. So you get a ticket for them. But you could get a ticket for almost any service you want in this way.

And their response is going to be this sort of interesting tuple. It's going to be your ticket between the client and the server encrypted just with that key KS as shown above. I guess we should write that down with KS

And also, you get to this shared key-- key CS. And this whole thing is encrypted with KC. So that's the wire protocol.

So I guess let's try to figure out a couple things. So first of all, how does the Kerberos server authenticate the client here? How does it know that this is the right user making this request? Yeah.

STUDENT: It can make sure that the ticket that it sent because it has KC.

PROFESSOR: Yes, I think that's what's going on is that the Kerberos server again on some level actually doesn't know whether this is the right client or not. But it thinks oh, well,

sure, it doesn't matter who is making this request. I'll just send this blob out and the only person who should be able to make any use of this blob is the person that knows this key KC over here. So that's actually kind of cool because the client doesn't have to send their password over the network at all.

So in some ways, this is actually better than the client sending a password to the Kerberos server because even if the Kerberos server here was listening for these passwords and trying to record them, it would never get your password. Or maybe if someone was impersonating the Kerberos server, they wouldn't get a copy of your password. All right, yeah.

STUDENT: [INAUDIBLE] adversary wants to [INAUDIBLE] your password offline without--

PROFESSOR: Yeah, so this is actually not a great aspect of Kerberos, in fact, right? So does everyone see what the problem is? The problem is that the way the client could tell if they got the right password or not or the workstation tells if the client supplied the right password is they try to decrypt this ticket and they see if it works or not.

And decryption is fairly cheap. This is symmetric encryption and you can do probably millions of decryptions a second if you try hard on modern machines. And this means that you can try millions of potential passwords per second to guess what the person's password is. And you could do this for any person at all. You could just send their principal to the Kerberos server.

It'll very happily give you back this response encrypted with the user's password. Then you can just try different passwords and just see what works or what doesn't. Yeah.

STUDENT: But won't the content [INAUDIBLE] decrypted [INAUDIBLE] advantage? How can we be sure that you directly--

PROFESSOR: Yes, this is actually another interesting aspect where the Kerberos 4 developers didn't quite realize that the time they were building this that they really should have been very careful about separating encryption from authentication. So in the paper, there's this implicit assumption that-- hopefully, that's not us. All right, sorry.

So in the paper, there's this implicit assumption that whenever you encrypt a piece of data and you send it to someone else, if that person can decrypt the data and it sort of looks OK, then no, they must have gotten the right key and the data wasn't tampered with in flight. But it seems like a totally bad plan now that we think of it 30 years later. But at the time, it wasn't so clear.

So in order to do Kerberos right, and in fact, what Kerberos 5 does now, is they both encrypt all the pieces of data and they also authenticate the message by basically computing a hash with a key. And then the result actually tells you that, oh, that piece of data just wasn't tampered with. It was correctly signed with this key, et cetera. And what actually happens in Kerberos version 4 is there are some extra bits in this thing that was encrypted that should all be some pattern like zeros.

And typically, if you get the key wrong, that pattern will not look like all zeros just by chance. It's not cryptographically guaranteed to be that. But most times, it will not look like zeros and you will be able to decide whether you got the correct key or not.

All right, so that's sort of the plan for how the clients tells, I guess, whether the ticket is valid. They just try to decrypt it and see how it works. So another interesting question is why is this key KCS included twice in the ticket in some form, right? So it's included once here and another time actually sort of implicitly in this ticket T. Why do we have two copies of the same key KCS? Yeah.

STUDENT: The client can't decrypt that ticket because it's encrypted with a service key.

PROFESSOR: Yeah, so it's actually kind of cute, right? Like, there's this key that the client can get to. But then there's another copy of it in here. It's encrypted with KS.

And the reason for this is that the Kerberos server is actually trying to set up the client and this other guy to talk to each other securely. So the Kerberos generates this hopefully random key KCS and wants to give one copy to the client and one copy to the write other server that you want to talk to. And one thing you could imagine doing naively is maybe the Kerberos will just go and say, hey service, this

guy wants to talk to you. Here's the key for it.

But that would be kind of unfortunate. You'd have to have the Kerberos server call back to the service and so on. So instead, these guys have this nice trick where they just to give the client does blob that the client can't actually do anything with other than give to the right service. And if the service has the right key KS, they'll decrypt it and say, aha. Well, here's the key I should be using to speak to this client. And that's how these two guys, the client and the service, are going to establish a shared key for protecting their communication. Yeah.

STUDENT: So what exactly is TGS?

PROFESSOR: So TGS is-- OK, so there's sort of two sides to it. From the client's point of view, it's just another service that you can get a ticket for. And the kinds of operations it supports is getting more tickets. It's a Ticket Granting Service.

STUDENT: Sorry, I meant what is the ticket called TGS.

PROFESSOR: Oh, yeah, sorry. This TGS is just shorthand for this whole blob except where S is actually the principal name of this TGS service. So you can think of it as like, well, there's a Kerberos server, there's this TGS service out there, and then there's the real thing I want to get to. So you first ask this guy to give me a ticket for some service.

You could ask it to give you directly a ticket for the file server. And this would work. But you'd need your KC to decrypt it and then you'd need your KC around all the time.

So instead, what you do is you get a ticket for this special service over here. It looks just like a service except that it happens to be provided by the same box. And then this guy will happily give you more tickets later without having to present your initial KC again.

Makes sense? All right, other questions? Yeah.

STUDENT: So [INAUDIBLE] the idea is once you get the TGS ticket, you can just get rid of your

KC?

PROFESSOR: Yes, so that's actually the cool thing about it is that once you get this ticket-- well, this ticket with the S being TGS, then you're going to get rid of the password and KC. So you log into Athena workstation and a couple of seconds into the boot process, you already get your ticket here. It scrubs the password from memory.

So even if someone, like, grabs you and grab the machine and runs off, all they got was your ticket. And OK, well, maybe they can access your stuff for 10 hours or whatever the ticket lifetime was, but not for longer than that. The password is gone. Yeah.

So if the password's gone, then on that picture there when Kerberos sends a reply encrypted with KC, how does the client--

Oh yeah, so this is the one place you need your password. So you send this message, you get this reply, you decrypt this, and then you forget the password. So you can't forget about the password before you use it for decryption of course.

STUDENT: [INAUDIBLE]

PROFESSOR: Ah, no, so this is just this interface, right? So this is the thing you do initially. And we'll talk in a second about how you basically also can get any ticket you want from the second interface without needing the initial key KC.

Right, make sense? All right, so I guess we already talked about two particular problems that the Kerberos protocol had sort of baked into it, which is a little unfortunate. One is that they assumed that encryption also provides authentication or integrity of messages. So don't do that. And Kerberos version 5 fixes this by explicitly authenticating messages. Another thing they sort of had a problem with is the ability for arbitrary clients to guess people's passwords.

So any suggestions of how we could fix this? How do you prevent guessing attacks in a protocol like this? What could we try? Yeah.

STUDENT: Some sort of salting? I'm not sure.

PROFESSOR: Well, so salting would just means that the client has to hash the password in different ways, maybe. But it still doesn't prevent them from trying lots of things. So maybe it'll be more expensive to build a dictionary. Yeah.

STUDENT: You could [INAUDIBLE] derivation function?

PROFESSOR: Yeah, so another good idea is to make this hashing process super expensive. So that might be kind of nice, right? So if this hash function took a second to compute like you guys did in lab two, then OK, wow, this would be actually really expensive to try different passwords. So that seems like a reasonable plan. So in combination with salting, those would be make it pretty expensive to do password guessing attacks. Anything else?

So another thing is, yeah, challenge respond. So you could actually hear in the initial protocol, the Kerberos server doesn't have any idea if this was the right client or not.

But in fact, what you could do is maybe give a little bit of a proof that, well, you're probably the right client. So maybe you could encrypt the current time stamp with your hash password or something like this-- has them together. And then the Kerberos server could just check if that's the right-- if that matches, and if so, return you back a ticket.

You probably don't want to necessarily add more rounds but this could work. So just to precise about what I'm sort of suggesting. OK, well, maybe you take the current time stamp and maybe you hash the current time stamp and the KC together. And maybe you include the timestamp as well.

And then the server could see, well, it has your KC. It could hash the current timestamp as well. If it gets the same value, then yeah, it's probably the right user requesting it.

And I can send back the ticket. If not, then it wasn't the right password at all.

Question?

STUDENT: [INAUDIBLE] you just do [INAUDIBLE] if the servers see too many requests [INAUDIBLE]

PROFESSOR: That's right. So the problem is that we could write limit. But there's no reason for the hacker to request this more than once.

That hacker requests a particular user more than once and then it gets this encrypted blob. And then it can try decrypting it offline as many times as it wants with different passwords without having to re request it. So I think the whole point of including some sort of a challenge response thing like this in the particle is so that the server will-- you'll have to actually ask the server again and again to try to log in with different passwords. And then you could rate limit of the server and get a much better defense. Yeah.

STUDENT: [INAUDIBLE] Kerberos?

PROFESSOR: So I think you could certainly replay this message so if I sent this message now, you could probably look at that message and send it as well and get a response back from the Kerberos server. I guess if you're watching the network, you could observe this thing on the wire as well. So I think this is sort of a bit of a stopgap measure-- improves security a bit.

But certainly if you're watching someone else's network, then you're going to see this packet coming back regardless of what happened in this step. So coming back, you'll see this guy and you can then try to attack it. There's probably some more elaborate schemes you could design but I didn't think Kerberos 5 even implements anything more elaborate than roughly this plan, which seems good enough to prevent arbitrary people from trying to break anyone's or brute force anyone's password. Make sense? Yeah.

STUDENT: So presume that you could do authenticated [INAUDIBLE] or something here to establish the shared key. And then you could encrypt this thing with KC and the shared key.

PROFESSOR: That's right, yeah. So if you're really doing this right, there's all these nice particles out there that are basically called password authenticated key exchange particles, which is exactly what's going on here. So if you're actually building a system, you should basically Google for SRP or PAKE. And these protocols and related particles will actually do this in a much better way where you can prove to both parties that you established a new key and both parties are convinced that it's the right other party and there's no way to mount these offline password guessing attacks on the set of network packets that you observe and so on.

So these are the sort of protocols. And they're much more elaborate in terms of crypto they rely on. So it's hard to explain on a board exactly why they work. Yeah.

STUDENT: [INAUDIBLE] part of the reason they did it this way is because they wanted to maintain the ability of just sending the password. And protocols just allow you to send a single thing as your authentication [INAUDIBLE].

PROFESSOR: Well, yeah, there's lots of sort of weird requirements that these guys had in mind. I think they-- well, certainly in practice, these servers could accept both Kerberos and non Kerberos connections. And for non Kerberos connections, you get-- like someone connects to the mail server but they're not using an Athena workstation.

They just want to send their password. And then the mail client here, let's say, is going to take your password and is going to get a ticket on your behalf just to check it. And then it's going to allow you to use it. So you certainly want conversion from Kerberos from passwords into checking against Kerberos. I don't think this precludes it because certainly Kerberos 5 deploys although this hashes of timestamps, et cetera.

STUDENT: Yeah but it's because they wouldn't want multiple [INAUDIBLE].

PROFESSOR: Yeah, well, I think that probably doesn't matter quite as much. You could certainly have multiple rounds in the back end behind your library. But there's some downsides to these particles-- probably not significant enough to stop you from using them. Other questions?

All right, I guess the other thing I want to mention that you should watch out for in the paper is that these guys, in designing Kerberos 4, they picked a single encryption scheme. And at the time, they basically picked DES, which was a popular encryption scheme of the time. It's a symmetric block cypher.

It goes pretty fast. It was reasonably secure, not necessarily the best, but certainly good enough at the time. And they just baked it into the protocol.

Everything in Kerberos has to use single DES or at least everything in Kerberos version 4. And this was a bit problematic because 25 years later, 30 years later now, it's actually very cheap to brute force DES encryption because the keys are actually very small. They're 56 bits.

So you could just search build some custom hardware that iterates over all the possible 2 to the 56 combinations and tries them all and figures out what someone's password is. So this is something also you want to avoid in any protocol you design now. Kerberos version 5 actually supports multiple different encryption schemes including AES and other things as well. So that seems like a much better way to do it.

On the other hand, MIT actually kept supporting DES up until two years ago, which is a little unfortunate. But now they don't. So that's good. Your principal is secure at least from this kind of attack.

All right, so does that make sense? This is the initial way you'd get any ticket at all in Kerberos. And typically, you'd get this ticket from this TGS service.

So now let's look at what's going on in this TGS service. So here, the interaction with the TGS service is going to be a little different. On one hand, you're going to-- as a client, you're going to have to speak to it as if you're speaking to any other Kerberos enabled service.

So we'll see how you authenticate yourself with a ticket to some machine. But then the response you're going to get back is just a ticket for some other principle that

you're going to want to communicate with like your file server. So the protocol level messages that show up here kind of look like this.

So here's your TGS service. And here's a client. The client already got a ticket for TGS using this protocol above. So what the client is actually going to send over is some combination of messages that prove that this is the right client and they're issuing a request for some particular principle through TGS.

So what the client is going to send to TGS is this tuple. So first, it's going to say, well, here's the service that I want to talk to next. So this might be your mail server or your file server. Then is going to include the ticket. It already got 4TGS.

So this is going to be TC of TGS encrypted with KTGS. So this is just this thing where S is TGS. And then you're going to have to include this authenticator blob. This is this AC thing from up there.

And this thing is going to be encrypted with a shared key between the client and the TGS service. So this is the message that you're going to send to TGS. It's going to look at this message, do something with it that we'll talk about in a second, and respond back with a ticket for this new service S. So the response here looks almost exactly like here. In fact, it is exactly the same thing.

It's going to be a ticket between the client and this new service S encrypted with KS and the shared key between the client and this new service S encrypted-- well, now, here's a little bit different. Instead of encrypting with KC, which the client has probably forgotten since then, now we're going to encrypt it with this shared key between the client and the TGS service. Makes sense?

All right, so in this-- how does the server actually figure out what the client wants to do? Or, how does server authenticate the client? Well, in this case, it's going to-- TGS server actually knows its own key, KTGS. So it's going to first decrypt this blob and look inside the ticket and figure out what's going on.

And there's all those nice fields in the ticket. So let's just double check. Why do we need all those fields in the ticket?

So is it important to have the server name S in the ticket? What would go wrong if you didn't have S in there? Anything? Yeah.

STUDENT: They could potentially get authorized to use any server.

PROFESSOR: Yeah, so it's in general a good idea to be very explicit in network protocols and to say exactly what a message means. So in this case, if you omit an S, you might be relying on the fact that, well, if it's the wrong S that you're trying to use the ticket for, then maybe you'll have a different key over here and then it wouldn't decrypt or something like this. But it seems like a good idea to include it to make sure that the server that receives this tickets decrypts and checks. Is that a ticket for me or for someone else? Yeah.

STUDENT: What does the client get KTGS on?

PROFESSOR: Ah, good question. The client has no idea what this is. Because this is like a super secret key. If you knew this, you'd probably be able to break all of Kerberos. So the client has no idea what KTGS is.

STUDENT: [INAUDIBLE]

PROFESSOR: Ah, yeah, yeah. And then the where you get it from is actually-- it's the Kerberos server itself that generates this whole blob for you where this is actually TGS and this is KTGS over here. So you don't construct this yourself. You just copy it over. OK

So what is the client name there important for? That should be fairly easy, right? If you don't put the client name in the ticket, then the server gets this nice blob but it has no idea who it's trying to talk to.

So it has no idea whether it should issue a ticket for my principal or for someone else's principle, et cetera. So what's the deal with the other fields? Why do these guys stick an address in the ticket?

This is the client's IP address. Does it matter? Yeah.

STUDENT: Can they use that address with the ticket to verify against the [INAUDIBLE]?

PROFESSOR: Sort of. Well, so I think the general plan for why there's addresses appearing everywhere here, these IP addresses, is that these guys at the time were still slightly confused and-- well, in some ways-- they were still thinking, OK, well, we're going to rely on IP addresses for a little bit of security. So they wanted to make sure that if the client logged in from some IP address, then everything else going on with that ticket happen from that same IP address.

So if you logged in from some IP address on 18.26.4.9 or something, then every connection you make to a file server or to a mail server has to be from the same IP address. Otherwise, the server should reject your connection as being stolen by-- or someone stole your ticket. So we're thinking, OK, well, maybe we'll defend against ticket theft this way.

If you still a ticket, well, but you're not using the same IP address. So it won't work. It's probably a little bit misguided at this point but-- that sort of gets in the way And Kerberos 5 still has it but it's largely optional. Really, you should just rely on cryptography instead of any IP address security.

So what's the point of the time stamp and lifetime things in the ticket up there? One are those guys good for? Are they useful? Yeah.

STUDENT: Preventing replay attacks.

PROFESSOR: Well, so the syndicator is the thing that's going to help us prevent replay attacks in a second. Because that thing gets generated every time you do a new request. On the other hand, the ticket just stays the same. So it's certainly not preventing replay attacks there. Yeah.

STUDENT: It prevents somebody from stealing your ticket then using it [INAUDIBLE]

PROFESSOR: Oh, sorry, yes. This just bounds the time for which a ticket is valid, meaning that the damage from disclosing it is hopefully reduced. So the plan is the timestamp is

roughly the time when you initially got the ticket.

And a lifetime in the ticket represents how many hours, let's say, it's valid from that initial timestamp. So if you try to use it too early or too late, then every server should reject such a ticket in the Kerberos protocol. So this kind of means that every server has to have a loosely synchronized, clock which is a bit of a-- well, maybe you've run into this.

Like, your laptop clock is off and you can't log into Kerberos anymore. Question? Yeah.

STUDENT: You said before that the client discards KC but they're still keeping KCS [INAUDIBLE] the TGS. [INAUDIBLE]

PROFESSOR: That's right, yeah. So the client discards KC after logging in. But it still keeps KCS. You're exactly right.

STUDENT: So if someone steals the KCS, then they have access to [INAUDIBLE].

PROFESSOR: Yeah, OK. So how bad is that? Like, why is it better to disclose this KCS fof-- actually, well, this is TGS, right? Why is it better to disclose KCTGS than KC? Yeah.

STUDENT: Someone [INAUDIBLE] somewhere [INAUDIBLE].

PROFESSOR: It's the kind of thing that they're both keys, though. So neither of them are really hashed, yeah.

STUDENT: You would take KCS and you'd just steal that session between those two. But if you steal KC, you can impersonate the client.

PROFESSOR: That's right. Yeah. So I guess one way to answer this is that KCTGS, this is actually a new key generated every time you log in initially. And this thing is only good because you have this ticket that goes along with it. If you lose this ticket or if this ticket is no longer valid, then yeah, you have these 56 bits in this key.

But no one is going to assume anything from those bits. The only reason these bits

are interesting is because this ticket talks about this KCS being valid right now. And there's a bound on it.

STUDENT: Yeah, so if they stole both of those [INAUDIBLE] be bounded.

PROFESSOR: Yeah, if someone steals both of these blogs, than they can impersonate you or, like, log into your file server, mail server for the lifetime of that ticket, which is a couple of hours or 10 hours. Stealing this, there's no time bound on that until you change your password and maybe worse. Make sense?

All right, so it seems like, yeah, all those fields are kind of important, IP address maybe less so. And now in response, right, we can get this ticket finally. And because we know KCTGS, we can decrypt the response from this TGS server. And now we have a ticket for any server we want-- a file server, mail server, whatever it is that we finally care about connecting to. Make sense?

All right, so let's look at how you might sort of finally use this in some application level protocol. So suppose that maybe I'm talking to a mail server to fetch my messages. So presumably, what my client workstation is going to do is going to send a tickets requesting for, I don't know, mail.po12.

And it'll get back a ticket for the principal mail.po12 or something like this. And then inside of this ticket or inside of this response, now I have a shared key between me and the mail server-- S is the male server over here-- and this ticket a blob that I can to the mail server to convince it that I'm the right guy or anyone with this key is the right principle. And then we can actually have an encrypted conversation with the mail server using this new key KCS.

So what I might do as a client is-- well, initially, I send some message to the mail server that includes this ticket TC mail encrypted with the key of the mail server. And then I can actually send some message along with this request that maybe says something like, well, delete some message-- delete 5. And I can encrypt this with KC mail.

Does that make sense? OK, so what happens in this protocol on the mail server

side? The mail server is going to use its secret key K mail to decrypt this ticket first.

And then it looks inside there and finds two important things-- the principal name C of who is it that's talking to it in the first place and the key KCS that it should be using to decrypt all the subsequent traffic and authenticate it ideally in Kerberos 5, at least. And then you can decrypt this message and say oh, well, yeah. User C is trying to delete message five. So I'll run this command. Make sense? You had a question?

STUDENT: Yeah, so Kerberos initially sends the TGS ticket in KCTGS. Where's [INAUDIBLE]?

PROFESSOR: So AC, those authenticators are actually generated by the client. Note that the client only needs KCS to generate an authenticator. So the client can make these up any time it wants.

So the general plan for indicators or the reason to use authenticators is roughly to prevent replay attack. So the client, or at least in the way that the Kerberos 4 developers were intending it, the client, every time it sends a new request , it would generate a new authenticator to say OK, well, this is a new request. I'm issuing it now. It's different from all the previous requests. Go do it.

And the general plan was that the server would keep a cache of these authenticators that were sent within the last five minutes or so. So if it ever sees a duplicate authenticator, it says oh, that's a replay request. I'm going to reject it. And if it sees an authenticator that's outside of a five minute boundary, it doesn't have it in the cache.

But it will look at the time stamp in the authenticator and say, well, this is a very old authenticator. I'll just reject your request because it's too old. Send it again if you really care.

So that's the general plan for indicators. As with many things in Kerberos, they were slightly broken-- in Kerberos 4, at least. Because this authenticator actually says nothing about the message you're sending, right? It's some blob.

So the way you would use it, for example, in this mail server protocol is-- or at least in Kerberos 4-- well, you would generate some authenticator and you would be to take the authenticator and you would encrypt it with also KC mail. And the mail server would keep track of, well, yeah, you've sent this [INAUDIBLE] indicator before. No, you haven't.

But there's nothing here that connects the authenticator to the message you are sending. So for the first message, this was great. But when you send a second message, you're going to generate a second authenticator.

And someone on the network can say, oh, yeah, I got your new authenticator. I can take your new authenticator and splice in the old delete message. So I'll force you to delete the fifth message twice, even though the second command meant to send some other operation.

So Kerberos 5 gets this right where you actually stick something in the authenticator that relates to the command you're issuing. You could do this, of course, but sort of took a while for people to realize that, well, here's how you should design a protocol correctly. Make sense? Yeah, other question.

STUDENT: [INAUDIBLE]

PROFESSOR: Ah, so the client gets case email from this response. So the client, when it wants to talk to the mail server, it's going to ask the TGS for a ticket for the mail server. And here, S is basically this mail server's principal name.

So when it comes back, this says S equals mail and this server key S is equal to mail. And this KCS is actually KC mail. So this is how the client learns of the shared key that it has between it and the files and the mail server here. And there's a copy of it inside the ticket. Question back there?

STUDENT: How does the mail server get KC mail?

PROFESSOR: Ah, yes. So how does the mail server get this shared key? Like, the mail server might have never heard of your connection before, never heard of you. Where does

KC mail come from on the mail server side? Yeah.

STUDENT: Isn't it part of the ticket?

PROFESSOR: Yeah, yeah, so this is the cool thing. You send this ticket over to the mail server and the mail server knows its own secret key K mail. And it uses that to decrypt the ticket TC mail and the shared key is in there along with the name of whoever it is that you're sharing this key with.

That's how it finds out, oh, I'm talking to the guy and that's the shared key we should use. Makes sense? All right, so that's the sort of basic plan for how you use this protocol in some actual application.

There's-- well, there's a bunch of problems with this. So Kerberos is-- it's a nice paper to read but then there's all these problems these guys didn't know about 30 years ago. So it's sort of inevitable that there's problems you have to go through. So one interesting problem in the way Kerberos 4 encrypted and authenticated messages for applications is that they use the same key for encrypting messages from the client to the server as well as messages from the server back to the client. So suppose that the client issues, I don't know, a request to fetch a particular message. So then I say, you know, fetch the message 7. And I encrypt this thing with KC mail. That seems all great.

The mail server has the shared key that's going to decrypt this message. And it's going to send me back the body of this email message also encrypted with KC mail. Does anyone see a problem with this?

Why is this is potentially a bad thing to do? Anyone else? Sure.

STUDENT: So there's a chapter [INAUDIBLE] they can make [INAUDIBLE] look like some other things they want [INAUDIBLE]

PROFESSOR: Yes, so those are actually worrisome because I could send you any email message I want. So suppose I really want to delete some message that is sitting here in your inbox and I don't want you to read it. I know it's maybe message, I don't know, 23.

So I'm going to send you an email that says, delete 23. You're going to read it. You're going to fetch it and a response is going to come from the mail server saying, delete 23 encrypted with this key. And so far, it's not being sent to the mail server.

But if I look at the network at the right time and capture this packet, I can send the packet back to the mail server. It would look like a message saying delete 23 encrypted with the right key. And the mail server will say, oh yeah, sure.

You're trying to delete this message. I'll do it. So this is a bit of a problem because we are allowing an adversary to confuse the mail server into whether our message was generated by it or was sent to it in the first place. So this is quite troublesome.

So these are what's typically called in cryptography and protocol literature as reflection attacks. So you have any suggestions for how we can avoid this problem? Yeah.

STUDENT: Can't you just include a header saying its origins?

PROFESSOR: Yeah, so typically, you want to have some very unambiguous way to state what's going on. One way is to have a header in every message that says, this is going from the client to the server or from server to the client. And even better plan in practice turns out to be just use two separate keys.

Because you might want to have a long stream of data where you don't really have space for this header bit. So instead, what Kerberos 5 does is every time you establish a connection with some service , you actually negotiate two keys instead of just one key. And the first key is going to be used for encrypting stuff from client to server and other from server back to the client. So that seems like a much better way to do it in practice.

Make sense? All right, so I guess let's now talk a little bit about what happens with KDC. So the Kerberos server is pretty important to the system.

But what happens if this KDC goes down? So how bad is it to our system. Like, in

Athena, suppose if KDC crashes, does this affect your life? Well, if you use Athena.

STUDENT: Is that why you can't log in?

PROFESSOR: Yeah, so you can't log in. I guess the other thing is you also can't get tickets to new things as well. But kind of the cool thing is that the KDC is largely off the critical path for existing connection. So no data passes through the KDC.

And if you already have a ticket for something, you can keep using it and keep logging into some service over the network. So in that way, it's actually quite nicely cacheable. I guess the other nice thing these guys do is they actually have a way of replicating the KDC potentially.

So they have one master Kerberos server that stores the sort of primary copy of this whole database. And then they can have read only replicas that hold a copy of this database. They don't allow any updates to this like registering users or updating keys.

But they do allow responding to login and TJS requests. So this way, these backup clones of this Kerberos database allow you to keep logging in and keep talking to services even if the master crashed and hopefully make it possible to upgrade to master without breaking everything at the same time. Makes sense? Any questions? Yeah.

STUDENT: How hard is it to compromise the KDC server and [INAUDIBLE]?

PROFESSOR: Well, yes, this is a huge sort of target for any system that runs Kerberos. Because if you compromise this guy, you're in complete control of the system. You can mint tickets for any service you want, pretending to be client you want.

So this is pretty bad. So you really want to keep this guy secure. Now, how hard is it to compromise? Well, ideally, it's hard.

And I don't know of any instances where the MIT KDC has actually been compromised in, I guess, 20 years or so. So it's , I think, possible to run this reasonably security. But presumably, the things you would worry about are just

software implementation security of the things that's listening on these two services, right?

So if there's buffer overflows on this guys or some other vulnerability like that, that's really bad. Or if there's an SSH server running on the Kerberos KDC and someone guesses the root password on that SSH server, they'll just log in and copy the database over. So I think you really want to be careful in minimizing the attack surface there.

Maybe be very careful writing the KDC code. Don't allow you to log into it directly. Maybe you even worry about physical security, et cetera. Absolutely.

Luckily, this is actually one of the few places where you have to be super paranoid, though. Right, so the servers, unlike in some other systems that do trust all the machines, the servers are actually not that important. They of course store some data.

But if someone compromises a mail server or a print server, you can probably recover reasonably well. So actually, here's an interesting question. Like, suppose someone compromised the mail server.

What should you do to recover from this attack? Like, if someone stole your mail, I guess that's kind of unfortunate. But what do you do so that the attacker doesn't keep accessing your mail in the future? Yeah.

STUDENT: [INAUDIBLE]

PROFESSOR: Yeah, so in Kerberos, there's no sort of revoke operation. But what you could do is you could change generate a new key for the mail server and stick it in this database over here. And then you install a new mail server, give it the new key, and then some attacker that has the mail server's old key has no way of-- like, no influence at all on this mail server now, right? On the other hand, suppose you didn't change the mail server's key, K mail. How that is that?

STUDENT: [INAUDIBLE] fine.

PROFESSOR: OK, so suppose you didn't change the mail. You, like, install the new mail server. You patch whatever bug that hacker exploited.

But it still has the same key K mail. And maybe it's taken a day so all the tickets are expired. Can that hacker do anything interesting in the system anymore? Yeah.

STUDENT: [INAUDIBLE]

PROFESSOR: Yeah, OK, so you give the new mail server the old K mail. Is that bad? Yeah.

STUDENT: This is--

PROFESSOR: Sure.

STUDENT: [INAUDIBLE] mail server. And [INAUDIBLE] mail server because you can encrypt that initial ticket.

PROFESSOR: Right. So K mail is actually super important. And, OK, so you're saying you can decrypt all the things going on to the mail server. So suppose the client now connects to the mail server after it's been fixed up.

But the attacker still knows K mail from the last time they compromised the system. They can now decrypt this ticket k mail and they can now look inside the ticket to get the session key and they can use that to decrypt all the messages you send, all the responses you get back, and so on. So it's pretty important to change this key K mail.

And in many ways, it's actually even worse than just looking at the traffic. Because if the attacker know this key K mail, they can synthesize new tickets for the mail server without talking to the key DC. So suppose that I know K mail and I want to read your mail from the mail server. I'll just make up this ticket.

I'll plop down all those five fields in the right order. I'll generate a new key. I'll encrypt it with K mail.

It'll look like the real thing generated by the KDC. And I'll just connect to the mail server. And it'll decrypt it.

It'll decrypt correctly and then it'll think, oh, yeah, this is some particular user. And you can read all their mail and you know the shared key and so on. So it's critically important too that no one knows the secret key of a service.

Because otherwise, not only is the traffic to the service decryptable and observable, but also you could impersonate anyone to that service. So this is actually all pretty important in Kerberos. Makes sense? Any questions? Yeah.

STUDENT: So if the attacker has to [INAUDIBLE], what's stopping him from changing the [INAUDIBLE] key [INAUDIBLE]?

PROFESSOR: Yeah, so presumably, how you'd recover-- like, ISNT would, like, call up the guy that runs this KDC and say, oh man, our mail server got compromised. Why don't you go and, like, delete this key from there and put in this new key instead? So you'd probably want to have some out of bounds mechanism for proving that you're really the mail server.

Because we'll look at a second on how do you change keys-- like a password changing protocol, for example. And you can in general change passwords in Kerberos. So if you know the old password, you can change the user's password to a new password here. So in order to recover, you probably-- like the attacker, might get your key mail, change it to something else.

Someone with ISNT basically has to go to the Accounts office and say, hey, we're hear at ISNT. Could you please change the password of the mail server for us? And they're going to generate some new password the attacker doesn't know.

So yeah, otherwise, if the attacker knows this key K mail, there's nothing differentiating the attacker from you, from the real mail server operator. In fact, the attacker probably changed the key so now they know the new thing and you don't. It's like you're not on the mail server anymore. So absolutely, you need some out of band protocol for initially registering principles in the database and for changing

keys if you forget your password or someone changes for you and you lose it as well.

So there's someone at MIT or there's a group of people at MIT that help users register for accounts and change their passwords by, you know, you present your MIT ID and say, oh, well, OK. Well, whatever happened, we'll be able to change your key for you then. Make sense?

So it's pretty important, of course, to do that right so if the person that is allowing password resets does the wrong thing when checking your MIT ID, you'll be able to compromise the system as well, right? So they are sort of part of the trusted computing base in Kerberos, right? Like, anyone that can go and muck with the database is pretty important for security here.

All right, so let's look at another sort of interesting use of Kerberos, right? So you could use Kerberos to try to log into some remote machine over SSH. And the way this would work is, of course, very similar to a mail server.

You'd get a ticket for the SSH server and you'd send the ticket along with your SSH connection. But what if you're SSHing into Athena dot dial-up and you don't have a Kerberos client on your machine? You just want to log into Athena dot dial-up with your regular password.

So how would Athena dial-up authenticate you, then, if you're just plugging into this machine with a password? But you have no password for Athena dot dial-up. It's on a Kerberos server. So which should the dial-up machine do when you log into it with a password? Yeah.

STUDENT: You can access the [INAUDIBLE].

PROFESSOR: Yeah, so the dial-up is then going to basically play the same protocol logging you in. So it's going to send a request. This thing, right?

It's going to send a request to the Kerberos service asking, give me a ticket for, I don't know, this user Alice. And in response, it's going to get back this reply

encrypted with Alice's password. And then it's going to try the password you just applied and see if it decrypts correctly. And if it decrypts correctly, it's going to let you log in, right? Yeah.

STUDENT: You don't even really have to send your key to the SSH server. Because it could relay this back-- the dot encrypted thing with KC. It could relay that back to the user over the SSH connection.

PROFESSOR: Potentially, yeah. Right, so this requires some fancy SSH client that you might not have. But-- so yeah, absolutely right. If you want to do this right, you probably want to have a Kerberos client on your machine and get a ticket yourself or maybe proxy it somehow through the SSH server but not allow the SSH server to have your key. That's probably a good plan.

STUDENT: [INAUDIBLE] just, the server could get through this wall [INAUDIBLE]

PROFESSOR: That's right. Then you might decrypt it and send back. OK, but in either case, right, all we're doing here is someone is going to try to decrypt this blob with KC. And then the server is going to get this resolved and see if it looks right. It's going to allow you in.

Seem like a good plan? So, turns out this is actually a fairly dangerous thing to do and allows you to potentially log into the SSH server as anyone. And the reason is that previously, when we were talking about a client trying to log in, the client basically knew that it was trying to supply a legitimate password, it was getting a reply from the right Kerberos server, and if it can decrypt it, then probably the password works out correctly.

However, there's nothing here in this protocol that authenticates the fact that this reply is coming from the right Kerberos server. So if I try to log into a machine by typing in a password and the machine sends out this request and some response comes back that seems to be encrypted with the password I typed in, this response might not be from the Kerberos server. So suppose I have some machine I want to log into. I type a password X into it. And then the machine sends out this response.

And before the Kerberos server can respond back with the real reply, I'll send my own reply that looks like this real response encrypted with my password X. And the workstation to which I try to log in or the SSH server is going to decrypt it with my fake password. It's going to look OK because this response was generated by me rather than the real Kerberos server.

And you'll be able to log in This make sense? Why does this happen? Yeah.

STUDENT: [INAUDIBLE] there's no authentication from the Kerberos server [INAUDIBLE]

PROFESSOR: Right, yeah, so there's nothing really here that's tying this to the real Kerberos server. So the way that Kerberos fixes this for remotely accessible machines like Athena dot dial-up is that the dial-ups themselves have some sort of a secret key that they share with the KDC. So in order to log you in into a dial-up or to any workstation that really cares about checking whether you are really the right user, it's actually going to do two things.

It's going to first log you into Kerberos like this. But then just because this reply decrypts correctly, it's not going to trust that. It's going to try to get a service ticket for itself using TGS. So the dial-up machine here has its own secret key.

And it logs you in with this round one. Then it talks to TGS saying, oh, please give me a service ticket for my own principle, from the dial-up principle, for this client. Then it gets the response back.

And then it checks if it can decrypt the response correctly. Because it knows the dial-up key for this KS. And if this decrpyts, that it knows, oh yeah, I must have talked to the right Kerberos server. Because only the right Kerberos server would have sent me this second round ticket encrypted with my secret key K dial-up.

So this is actually pretty important. And typically, Athena workstations don't do this extra step because Athena workstations don't have any secret key stored on them that's shared with the KDC. Why is this OK for Athena workstations to let you log in in this one round trip and not for dial-ups? Yeah.

STUDENT: If you don't have access to any services because the attacker couldn't forge the ticket.

PROFESSOR: That's right, yeah. So there's nothing interesting on the dial-up machine itself-- sorry, on the workstation itself. So workstation, whenever-- you have root access on it anyway. So if you log into it with a fake password, who cares?

It's not like you'll be able to do anything else outside of your workstation. Whereas on a dial-up, things are much more interesting. It might be that you have other processes running on the dial-up from other login sessions. And there, the fact that you log in with a particular Unix UID is actually pretty important.

And there, they really want to authenticate that you are the right entity. So that's why they do this sort of 2-step process for logging into some shared time-sharing machine. Make sense?

All right, so the last thing I want to talk about is how do we change keys. So we sort of talked about it briefly here with the idea that the mail server's key might get compromised. But as a user, you probably also want to change passwords as well. Like, maybe you're thinking, oh, that password is not so great anymore.

Maybe I accidentally wrote it on a piece of paper and someone looked at it. So you probably want to change it. So the way this works is actually at some level fairly straightforward.

So there's an extra interface to this Kerberos server. In addition to Kerberos and TGS, there's this extra service called kpassword. And the service lets you change your password.

And the way it works is you get a ticket for this service very much like you'd get a ticket for the mail server or any other service. And then you send your new password to this kpassword service encrypted with your session key. And then if everything checks out, your key in the database is going to be updated to the new key. Question.

STUDENT: [INAUDIBLE] if there was, like, no [INAUDIBLE] they wanted to have them use a [INAUDIBLE] had to go through this.

PROFESSOR: That's right. OK, yeah, OK. So this is a good point.

So for changing your password, remember that we have this whole goal that if someone steals your ticket, it shouldn't be good enough to completely take over your account. So for this reason, the key password service actually doesn't accept just any ticket. It accepts a ticket that you initially get from the Kerberos service with your KC.

So the way this actually works is that inside of every ticket, in addition to all the stuff I showed you there, there's an extra bit that tells you which of these two things gave you the ticket. So if you get the ticket from this Kerberos server, the bit has one. If you get the ticket from the TGS server, the bit is zero, let's say.

And then the kpassword service, in addition to everything that any mail server or file server would do, it also looks at the bit on the ticket and says, well, if you got it from Kerberos, that's good. If you got it from TGS, that means that maybe you stole someone's ticket and you didn't know their password right away. So I'm not going to accept this.

So this is how a key password ensures that you can only change the password if you just knew the password immediately prior to this. So you never actually supply the old password to kpassword. You supply that to-- well, you supply that in order to decrypt the response from the Kerberos server for the kpassword password principal. Makes sense?

All right, so let's just actually spell out the interactions with the key password service because there'll be something a little bit interesting there. So when you're going to change your password, what the client is going to do is, of course, obtain this initial ticket from Kerberos. So it sends a message to the Kerberos service saying here is my client ID and I want to talk to the kpassword service.

And the Kerberos server is going to send a response back including the ticket

between the client and the kpassword service encrypted with a key of kpass and the shared key between KC and kpass. Makes sense? This is exactly this thing up here encrypted with KC.

Makes sense? Everyone's runs on board? And then very much like you talk to a mail server, you take this and you send it to kpassword. You say, well, here's my ticket-- tckpass encrypted with kpass. And in addition, you send your new password and you encrypt this with the key kcpass with shared key for your interaction.

I just separated these two things out here. Make sense? So this is the thing you send to the kpassword service with a new password encrypted with the session key. Yeah.

STUDENT: But in the [INAUDIBLE]

PROFESSOR: So if the attacker knows your password, they can change your password-- absolutely. So it seems reasonable, right? Like, there's no other way to tell whether it's you or not.

If someone walks up to an Athena workstation, types in your username and password, runs password, change my password to this new thing, they know your password? They're going to be able to change it. So that's totally the same in almost any system you could imagine.

This is true for Gmail probably. This is true for any system that uses passwords in general. The reason that we had to talk to the Kerberos server instead of the TGS server here is that if someone steals your ticket, then we don't want them to be able to change your password.

So if someone corrupts an Athena workstation after you log in, your password is gone from memory, the ticket remains. You could, in principle, use the ticket to obtain more tickets for the password changing service. But the password changing service says that's not going to be good enough. It's going to look at this ticket here-- ticket between the client and the password service.

And if that was updated through TGS, it's going to reject your request. It's only going to accept it if it was obtained directly from the Kerberos service using KC. Makes sense? Question.

STUDENT: So if you [INAUDIBLE] password with the [INAUDIBLE] Athena, right?

PROFESSOR: Yeah.

STUDENT: So if I would steal your private key--

PROFESSOR: Yeah, so actually, KC is basically equivalent to your password here. As far as the Kerberos protocol is concerned, that don't even think that you have a password. It thinks you have a private key KC. If someone knows KC, that's basically your password. So you can change your key from KC to something else so you don't lose this thing, yeah. Uh, yeah, question?

STUDENT: [INAUDIBLE] initial message [INAUDIBLE] from changing the [INAUDIBLE]

PROFESSOR: Oh, you can certainly get a ticket for some other service or an attacker could drop this message altogether or corrupt this exchange. And then you're not going to successfully change your password.

STUDENT: [INAUDIBLE]

PROFESSOR: Yeah, so the client actually knows exactly what service it wants to talk to. So if it-- it's going to make sure, well, there are some messages, some parts of this protocol I'm not showing that allow the client to make sure they actually got the ticket for the right thing. But yeah, so the-- sorry. Question?

STUDENT: [INAUDIBLE] very easy to denial of service attack when an attacker [INAUDIBLE] modifying the encrypted version of the new password.

PROFESSOR: Yeah, so there's actually lot of things. Because, for example, Kerberos doesn't do authentication properly of messages-- it just does encryption-- you could totally corrupt this blob. And--

STUDENT: [INAUDIBLE]

PROFESSOR: Yeah, you could flip it around and it'll decrypt to something else. The service was like, oh yeah, that's the new password, and set it. And then you're sort of locked out.

So it seems really unfortunate. And this is yet another reason why you want separate encryption and authentication in the protocol. There's actually an even bigger problem here. But actually, yeah, question?

STUDENT: What's the point of having a one way function to [INAUDIBLE] your password in KC if they're essentially [INAUDIBLE]?

PROFESSOR: Basically because KC is a 56-bit ds key and your password might be of different lengths. And even if it's longer than 56 bits, which is seven bites or seven characters, you want to use all those extra bytes too. So it's mostly just to condense it down to a fixed width blob, yeah.

But there's actually a much more interesting problem here, which is that suppose that I change my password and then I decide, OK, well, I change my password. A day goes by and I'm thinking, oh yeah, sure. I'll tell everyone what my stupid old password was. Is this a good idea in Kerberos? Yeah.

STUDENT: [INAUDIBLE] immediately expire until the--

PROFESSOR: Yeah, OK. But someone's got to wait for all my tickets to expire. I wait for, like, a week. And then no tickets are good anymore. Can I give out my old password now? Yeah.

STUDENT: It might take a while to replicate.

PROFESSOR: Yeah, suppose the replicas are all updated, yeah, all the stuff. Yeah.

STUDENT: [INAUDIBLE] if someone saves the initial transaction [INAUDIBLE] to get, like, your old password now that they have your new password.

PROFESSOR: Yeah, so this is actually super worrisome in Kerberos, which is that-- suppose some

attacker was watching all of my previous password changes. They didn't know what my password was or is or anything. But they're just saving these packets very diligently.

And then a month later, I go and say, oh, my password was poodle or something silly like this. And then they go say ah, ha, ha. I can now decrypt this initial thing because it was encrypted with your old KC. And I can get this KC with pass that you shared.

Then I can use this to decrypt the new password you sent to the KDC. And even if you change the password again, I can decrypt the next round as well. And you can just keep going and get the newest password.

So in this particular password, changing protocol, if you ever disclose an old password, then someone could sort of unzip this whole chain of encrypted messages and get your newest password as well. This is actually very troublesome in the design. Yeah.

STUDENT: Doesn't the later version of Kerberos [INAUDIBLE]

PROFESSOR: Absolutely, yes. So there's actually a solution to this that's not sort of fundamental. And this is something they didn't realize in Kerberos version 4.

There's actually this nice mechanism called Diffie-Hellman that I'll just sketch out in like one minute just so that you guys know when to use it or et cetera. But it's basically a solution for this kind of a problem where you want to stop this unzipping from happening. So what happens in the Kerberos version 5 password changing protocol is that actually you want to establish a new secret that will not be apparent if you happen to decrypt all the messages on the wire.

And the way this works is-- this is like some math that you don't have to fully understand. But the client some random value X. And the kpassword server is going to pick some other random value Y. And what they send to each other are exponentiations of these values. So the client sends G to the power X to the server and the server sends G to the power Y back to the client.

And it turns out that mathematically, what we can do now is the client can take G to the Y, raise it to X, and get this value G to the XY. The server can take G to the X, raise it to the power Y, and get G to the XY as well.

They can now use this secret value G to the XY to encrypt their subsequent messages, including the new password. So you send the new password encrypted with this value G to the XY, roughly speaking. But for some mathematical reasons that we're not going to cover right now, it turns out to be super difficult for someone that just gets G to the X and G to the Y by examining your packets later from figuring out what was G to the XY. So this is something called the discrete log problem. Yeah, question?

STUDENT: But they have to [INAUDIBLE] G at some point.

PROFESSOR: Yeah, yeah. So G is some parameter you could sort of send at the beginning of a protocol or it could be just cooked into Kerberos. It turns out to be relatively less important.

All right, so anyway, use Diffie-Hellman because this is called-- well, what you should Google for if you're building a protocol like this is this Diffie-Hellman key exchange protocol. And Kerberos 5 actually does this correctly. But this is something to really watch out for if you're designing any kind of new protocol yourself. All right, so that's it for Kerberos. Let's talk about SSL on Monday.