PROFESSOR: Now look at how the web uses cryptographic protocols to protect network communication and deal with network factors in general.

So before we dive into the details, I want to remind you there's a quiz on Wednesday. And that's not in this room. It's in Walker. But it's at the regular lecture time.

Any questions about that? Hopefully straightforward.

Third floor, I think it is usually.

All right. So today we're going to talk about how the web sort of uses cryptography to protect network communication. And we'll look at two sort of closely related topics.

One is, how do you just cryptographically protect never communication in a larger scale than the Kerberos system we looked at in last lecture? And then also, we're going to look at how do you actually integrate this cryptographic protection provided to you at the network level into the entire application.

So how does the web browser make sense of whatever guarantees the cryptographic protocol is providing to it?

And these are closely related, and it turns out that protecting network communications is rather easy. Cryptography mostly just works. And integrating it in, and currently using it at a higher level in the browser, is actually that much trickier part, how to actually build a system around cryptography.

Before we dive into this whole discussion, I want to remind you of the kinds of cryptographic primitives we're going to use here.

So in last lecture on Kerberos, we basically used something called symmetric crypto, or encryption and decryption. And the plan there is that you have a secret key k, and you have two functions.

So you can take some piece of data, let's call it p for plain text, and you can apply an encryption function, that's a function of some key k. And if you encrypt this plain text, you get a Cypher text c.

And similarly, there's a description function called d, that given the same key k. And the cipher text will give you back your plain text.

So this is the primitive that Kerberos was all built around.

But it turns out there's other primitives, as well, that will be useful for today's discussion.

And this is called asymmetric encryption and decryption. And here the idea is to have different keys for encryption and decryption. We'll see why this is so useful. And in particular, the functions that you get is, you can encrypt to a particular public key with a sum message and get a cipher text text. And in order to decrypt, you just supply the corresponding secret key to get the plain text back.

And the cool thing now as you can publish this public key anywhere on the internet, and people can encrypt messages for you, but you need the secret key in order to decrypt the messages. And we'll see how this is used in the protocol.

And in practice you'll often use public key crypto in a slightly different way. So instead of encrypting and decrypting messages, you might actually want to sign or verify messages.

Turns out that at the implementation level these are related operations, but at an API level they might look all little bit different.

So you might find a message with your secret key, and you get some sort of a signature s. And then you can also verify this message using the corresponding public key. And you get the message, and the signature, and outcomes, and some Boolean flags saying whether this is the correct signature not on that message.

And there are some relatively intuitive guarantees that these functions provide if you, for example, got this signature and it verifies correctly, then it must have been generated by someone with the correct secret key.

Make sense, in terms of the primitives we have? All right.

So now let's actually try to figure out--

How would we protect network communication at a larger scale in Kerberos. In Kerberos, we had the fairly simple model where we had all the users and servers have some sort of a relation with this KDC entity. And this KDC entity has this giant table of principles and their keys.

And whenever a user wants to talk to some server, they have to ask the KDC to generate a ticket based on those giant table the KDC has.

So this seems like a reasonably straightforward model. So why do we need something more? Why is Kerberos not enough for the web? Why doesn't the web use just Kerberos for securing all communications?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah. So there a sort of a single KDC, has to be trusted by all.

So this is, perhaps, not great. So you might have trouble really believing that some machine out there is secure for everyone in the world to use. Like, yeah, maybe people at MIT are willing to trust someone at [? ISNT ?] to run the KDC there.

All right. So that's plausible, yeah.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yes. A key management is hard, I guess, yeah. So what I mean in particular by key management--

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yes. It might actually be a hard job to build a KDC that we can manage a billion keys, or ten billion keys, for all the people in the world. So it might be a tricky proposition.

If that's not the case, then I guess another bummer with Kerberos is that all users actually have to have a key, or have to be known to the KDC.

So, you can't even use Kerberos at MIT to connect to some servers, unless you yourself have an account in the Kerberos database. Whereas on the web, it's completely reasonable to expect the you walk up to some computer, the computer has no idea who you are, but you can still go to Amazon's website protected with cryptography.

Yeah?

AUDIENCE: [INAUDIBLE]

PROFESSOR: That's our idea. So there's these kinds of considerations. So there's private forward secrecies. There are a couple of other things you want from the cryptographic protocol, and we'll look at them and how they sort of show up in SSL, as well.

But the key there is that the solution is actually exactly the same as what you would do Kerberos, and what you would do in SSL or TLS to address those guys.

But you're absolutely right. There Kerberos deep protocols we read about in the paper is pretty dated. So, even if you were using it for the web, you would want to

apply some changes to it. Those are not huge though, at the [INAUDIBLE] level.

Any other thoughts on why we should use Kerberos? Yeah?

AUDIENCE: [INAUDIBLE]

PROFESSOR: This is actually not so scalable.

Yeah, recovery. Maybe registration even, as well, like you have to go to some accounts office and get an account. Yeah?

AUDIENCE: [INAUDIBLE] needs to be online.

PROFESSOR: Yeah, so that's another problem. These are sort of management issues, but at the protocol level, the KDC has to be online because it has actually mediate every interaction you have with the service.

It means that in the web, every time you go to a new website, you'd have to talk to some KDC first, which would be a bit of a performance bottleneck. So like another kind of scalability, this is like performance scalability. This is more management scalability kind of stuff. Make sense?

So, how can we solve this problem with these better primitives? Well, the idea is to use public key cryptography to give this KDC out of the loop.

So let's first figure out whether we can establish secure communication if you just know some other party's public key. And then we'll see how we plug-in a public key version of a KDC to authenticate parties in this protocol.

If you don't want to use a KDC, what you could do with public key crypto is maybe you can somehow learn the public key of the public key of the other value on a connector. So in Kerberos, if I want to connect to a file server, maybe I just know the file server's public key from somewhere. Like me as a freshman I get a printout

saying the file server's public key is this. And then you can go ahead and connect it.

And the way you might actually do this is you could just encrypt a message for the public key of the file server that you want to connect to. But it turns out that in practice, these public key operations are pretty slow. They are several orders of magnitude slower than symmetric key cryptography. So almost always you want to get out of the use of public crypto as soon as practical.

So a typical protocol might look like this where you have a and b, and they want to communicate. And a knows b's public key. So what might happen is that a might generate some sort of session s.

Just pick a random number. And then it's going to send to b the session key s.

So this is kind of looking like Kerberos. And we're going to encrypt the session s for b's key. And remember in Kerberos, in order to do this, we have to have the KDC do this for us because a didn't know the key for b, or couldn't have been allowed to know because that is a secret. But only b should've known.

With public key cyrptor you can actually this now. We can just encrypt the secret s using these public keys. And we send this message over to b. B can now decrypt this message, and say I should be using this secret key.

And now we can have a communication channel where all the messages are just encrypted under this secret key s.

Does this Make sense?

So there are some nice properties about this protocol. One is that we got rid of having to have a KDC be online and generate our session key for us. We could just have one of the parties generate it and then encrypt it for another party without the use of the KDC.

Another nice thing is we're probably pretty confident that messages sent by a to b will only be read by b. Because only b could have decrypted this message. And

therefore, only b should have that corresponding secret key s.

But this is pretty nicely. Any questions about this protocol? Yeah?

AUDIENCE: Does it matter whether the user or the server generates the pass code?

PROFESSOR: Well, maybe. I think it depends on exactly the considerations, or the properties you want out of this protocol.

So here, certainly if a is buggy or picks bad randomness, the server then sends some data back to a, thinking, oh, this is now the only data that is going to be seen by a. Well, maybe that's not going to be quite right. So you might care a little bit. There's a couple of other problems with this protocol, as well.

Question?

AUDIENCE: I was gonna say that in this protocol, a you could just do [INAUDIBLE].

PROFESSOR: Yes, that's actually not great. So there's actually several problems with this. One is the replay.

So the problem here is that I can just send these messages again, and it looks like a is sending these messages to b, and so on.

So typically the solution to this is to have both parties participate in the generation of s, and that ensures that the key we're using is now fresh. Because here, because b didn't actually generating anything, these protocol messages look exactly the same every time.

So typically, what happens is that, one party picks a random number like s, and then another party b also picks some random number, typically called a non. But, whatever. There's two numbers. And then the key they agreed to use in the thing that one party picked, but actually is the hash of the things that both of them picked.

So you could do that. You could also do [? DP Helmond ?] kind of stuff like we looked at in the last lecture where you get forward secrecy. It was a little bit more

complicated math rather than just hashing two random numbers that two parties picked. But then you get some nice properties, like forward secrecy.

So replay attacks you typically fixed by having b generate some nons. And then you set the real secret key that you're going to use to hash of the secret key from one guy concatenated with this non. And, of course, b would have to send the nons back to a in order to figure out what's going on for both of them to agree on a key.

All right. So another problem here is that there's no real authentication of a here, all right? So a knows who b is, or at least a knows who will be able to decrypt the data. But b has no idea who is on the other side, whether it's a or some adversary impersonating a, et cetera.

So how would we fix it int his public key world?

Yeah?

AUDIENCE: You have been assigned something and [INAUDIBLE].

PROFESSOR: Yeah. There's a couple of ways you could go about this. One possibility is a maybe should sign this message initially, because we have this nice sign primitive. So we could maybe have a sign this thing with a's secret key. And that sign just provides the signature, but presumably you assign it and also provide the message, as well.

And then b would have to know a is public key in order to verify the signature. But if b knows a is public key, then b's going to be reasonably confident that a is the one that sent this message over.

Make sense?

Another thing you could do is rely on encryption. So maybe b could send the nons back to a encrypted under a's public key. And then only a would be able to decrypt the nons and generate the final session key s prime.

So there are a couple of tricks you could do. This is roughly how client certificates work in web browsers today.

So a has a secret key, so when get an MIT personal certificate, what happens is your browser generates a long lived secret key and gets a certificate for it. And whenever you send to request a web server, you're going to prove the fact that you know the secret key in your user certificate, and then establish the secret key s for the rest the communication.

Make Sense? All right.

These are sort of all fixable problems at the protocol level that are reasonably easy to V address by adding extra messages. The big assumption here, of course, that we're going under is that all the parties know each other's public keys.

So do you actually discover someone's public key? for, you know, it a wants to connect a website, I have a URL that I want to connect to, or a host name, how do I know what pub key that corresponds to? Or similarly, if I connect to websis to look at my grades, how does the server know what my public key should be, as opposed to the public key of some other at person MIT?

So this is the main problem that the KDC was addressing. I guess the KDC was solving two problems for us before.

One it that is was generating this message. It was generating the session key and encrypting it for the server. We fixed that by doing public key crypto now.

But we also need to get this mapping from string principal names to cryptographic keys of the Kerberos previously provided to us. And the way that is going to happen in this the https world, this protocol called TLC, is that we're going to still rely on some parties to maintain, of to a least logically maintain those giant tables mapping principal names onto cryptographic keys.

And the plan is, we're going to have something called a certificate authority.

This is often abbreviated as CA in all kinds of security literature. This thing is also

going to logically maintain the stable of, here's the name of a principle, and here's the public key for that principle.

And the main difference from the way Kerberos worked, is that this certificate authority thing isn't going to have to be online for all transactions.

So in Kerberos you have to talk to those KDCs to get a connection or to look up someone's key.

Instead, what's going to happen in this CA world, is that if you have some name here, and a public key, the certificate authority is going to just sign messages stating that certain rows exist in this table.

So the certificate authority is going to have its own sort of secret and public key here.

And it's going to use the secret key to find messages for other users in the system to rely on. So if you have a particular entry like this, in this CA's database, then the CA is going to find a message saying this name corresponds to this public key. And it's going to sign this whole message with CA's secret key.

Make sense?

So this is going to allow us to do very similar things to what Kerberos was doing, but we are now going to get rid of the CA having to be online for all transactions. And in fact, it's now going to be much more scalable. So this is what's usually called a certificate.

And the reason this is going to be much more scalable is that, in fact, to a client, or anyone using this system, a certificate provided from one source is as good as a certificate provided from any other source. It's signed by the CA secret key. So you can verify its validity without having to actually contact the certificate authority, or any other designated party here.

And typically, the way this works is that a server that you want to talk to stores the certificate that it originally got from the certificate authority. And whenever you connect to it, the server will tell you, well, here's my certificate. It was signed by this CA. You can check the signature and just verify that this is, in fact, my public key and that's my name.

And on the flip side, the same thing happens on client certificates. So when you the user connect to a web server, what's actually going on is that your client certificate actually talks about the public key corresponding to the secret key that you originally generated in your browser. And this way when you connect to a server, you're going to present a certificate signed by MIT's certificate authority saying your user name corresponds to this public key. And this is how the server is going to be convinced that a message signed with your secret key is proof that this is the right Athena user connecting to me.

Does that make sense? Yeah.

AUDIENCE: Where does the [? project ?] get the certificate [INAUDIBLE]?

PROFESSOR: Ah, yes. Like the chicken and the egg problem. It keeps going down. Where do you get these public keys? At some point you have to hard code these in, or that's typically what most systems do.

So today what actually happens is that when you download a web browser, or you get a computer for the first time, it actually comes with public keys of hundreds of these certificate authorities.

And there's many of them. Some are run by security companies like VeriSign. The US Postal Service has a certificate authority, for some reason. There's many entities there that could, in principal, issue these certificates and are fully trusted by the system.

These mini certificate authorities are now replacing the trust that we had in this KDC.

And sometimes, we haven't actually addressed all the problems we listed with Kerberos. So previously were worried that, oh man, how are we going to trust? How is everyone in the word going to trust a single KDC machine?

But now, it's actually worse. This is actually worse that in some ways, because instead of trusting a single KDC machine, everyone is now trusting these hundreds or certificate authorities because all of them are equally as powerful. Any of them could sign a message like this and it would be accepted by clients as a correct statement saying this principle has this public key. So you have to only break into one of these guys instead of the one KDSC.

Yeah?

AUDIENCE: Is there a mechanism to open the keys?

PROFESSOR: Yeah. That's another hard problem. It turns out to be that before we talked to the KDC, and if you screwed up, you could tell the KDC to stop giving out my key, or change it. Now the certificates are actually potentially valid forever.

So the typical solution is twofold. One is, sort of expectedly, these certificates include an expiration time. So this way you can at least bound the damage. Is This is kind of like a Kerberos ticket's lifetime, except in practice, these tend to be to several orders of magnitude higher. So in Kerberos, your ticket's lifetime could be a couple hours. Here it's typically a year or something like this.

So the CAs really don't want to be talked to very often. So you want to get your money once a year for the certificate, and then give you out this blob of signed bytes, and you're good to go for a year. You don't have to conduct them again.

So this is good for scalability, but not so good for security.

And there's two problems that you might worry about with certificates. One is that Maybe the CA's screwed up. So maybe the CA issued a certificate for the wrong name. Like, they weren't very careful. And accidentally, I ask them to give you a

certificate for amazon.com, and they just slipped up and said, all right, sure. That's amazon.com. I will give you a certificate for that.

So that seems like a problem on the CA side. So they miss-issued a certificate. And that's one way that you could end up with a certificate that you wish no longer existed, because you signed the wrong thing.

Another possibility is that they CA does the right thing, but then the person who had the certificate I accidentally disclosed the secret key, or someone stole the secret key corresponding to the public key in the certificate. So this means that certificate no longer says what you think it might mean. Even though this says amazon.com's key is this, actually every one in the world has the corresponding secret key because posted it on the internet.

So you can't really learn much from someone sending you a message signed by the corresponding secret key, because it could've been anyone that stole the secret key.

So that's another reason why you might want to revoke a certificate. And revoking certificates is pretty messy. There's not really a great plan for it. The two alternatives that people have tried are to basically publish a list of all revoked certificates in the world. This Is something called certificate revocation list, or CRLs. And the way this works is that every certificate authority issues these certificates, but then on the side, it maintains a list of mistakes.

These are things that it realized they screwed up and issued a certificate under the wrong name, or our customers come to them and say, hey, you issued me a certificate. Everything was going great. But someone then got rude on my machine and stole the private key. Please tell the world that my certificate is no good anymore.

So this certificate authority, in principle, could add stuff to this CRL, and then clients like web browsers are supposed to download this CRL periodically. And then whenever they're presented with a certificate, they should check if the certificate

appears in this revoked list. And it shows up there, then should say that certificate's no good. You better give me a new one. I'm not going to trust this particular sign message anymore.

So that's one plan. It's not great.

If you really used, it would be a giant list. And it would be quite a lot of overhead for everyone in the world to download this. The other problem is that no one actually bothers doing this stuff. so the lists in practice are empty. If you actually ask all these CAs, most of them will give you back an empty CRL because no one's ever bothered to add anything to this list.

Because, why would you? It will only break things because it will reduce the number of connections that will succeed.

So it's not clear whether there is a great motivations for CAs to maintain this CRL.

The other thing that people have tried is to query online the CAs. Like in the Kerberos world, we contact the KDC all the time. And in the CA world we try to get out of this business and say, well, the CA's only going to sign these messages once a year. That's sort of a bummer. So there's an alternative protocol called online certificate status protocol, or OCSP. And this protocol pushes us back from the CA world to the KDC world.

So whenever a client gets a certificate and they're curious, is this really a valid certificate? Even though it's before the expiration time, maybe something went wrong. So using this OCSP protocol, you can contact some server and just say, hey, I got this certificate. Do you think it's still valid? So basically, offloading the job of maintaining this CRL to a particular server.

So instead of downloading a whole list yourself, you're going to ask the server, hey, is this thing in that list? So that's another plan that people have tried. It's also not used very widely because of two factors.

One is that it adds latency to every request that you make. So every time you want

to connect to a server, now you have to first connect, get the certificate from the server. Now you have to talk to this OCSP guy and then wait for him to respond and then do something else. So for latency reasons, this is actually not a super popular plan.

Another problem is that you don't want this OCSP thing being down from affecting your ability to browse the web. Suppose this OSCP server goes down. You could, like, disable the whole internet because you can't check anyone's certificate.

Like, it could be all bad. And then all your connections stop working. So no one wants that. So most clients treat the OCSP server being down as sort of an OK occurrence.

This is really bad from a security perspective because if you're an attacker and you want to convince someone that you have a legitimate certificate, but it's actually been revoked, all you have to do is somehow prevent that client from talking to the OCSP server. And then the client will say, well, I do the certificate. I'll try to check it, but this guy doesn't seem to be around, so I'll just go for it. So that's basically the sort of lay of the land as far as verification goes. So there's no real great answer.

The thing that people do in practice as an alternative to this is that clients just hard code in really bad mistakes. So for example, the Chrome web browser actually ships inside of it with a list of certificates that Google really wants to revoke. So if someone mis-issues a certificate for Gmail or for some other important site-- like Facebook, Amazon, or whatever-- then the next release of Chrome will contain that thing in its verification list baked into Chrome.

So this way, you don't have to contact the CRL server. You don't have to talk to this OCSP guy. It's just baked in. Like, this certificate is no longer valid. The client rejects it. Yeah.

AUDIENCE: Sorry, one last thing.

PROFESSOR: Yeah.

AUDIENCE: So let's say I've stolen the secret key on the certificate [INAUDIBLE]. All public keys are [? hard coded-- ?]

PROFESSOR: Oh, yeah. That's [INAUDIBLE] really bad. I don't think there's any solution baked into the system right now for this. There have been certainly situations where certificate authorities appear to have been compromised.

So in 2011, there were two CAs that were compromised in the issue, or they were somehow tricked into issuing certificates for Gmail, for Facebook, et cetera. And it's not clear. Maybe someone did steal their secret key.

So what happened is I think those CAs actually got removed from a set of trusted CAs by browsers from that point on. So the next release of Chrome is just like, hey, you're really screwed up. We're going to kick you out of the sort of CAs that are trusted.

And it was actually kind of a bummer because all of the legitimate people that had certificates from that certificate authority are now out of luck. They have to get new certificates. So this is a somewhat messy system, but that's sort of what happens in practice with certificates. Make sense? Other questions about how this works?

All right. So this is the sort of general plan for how certificates work. And as we were talking about, they're sort of better than Kerberos in the sense that you don't have to have this guy be online. It might be a little bit more scalable because you can have multiple KDCs, and you don't have to talk to them.

Another cool thing about this protocol is that unlike Kerberos, you're not forced to authenticate both parties. So you could totally connect to a web server without having a certificate for yourself. This happens all the time. If you just go to amazon.com, you are going to check that Amazon is the right entity, but Amazon has no idea who you are necessarily, or at least not until you log in later.

So the crypto protocol level, you have no certificate. Amazon has a certificate. So that was actually much better than Kerberos where in order to connect to a Kerberos service, you have to be an entry in the Kerberos database already. One

thing that's a little bit of a bummer with this protocol as we've described it is that in fact, the server does have to have a certificate.

So you can't just connect to a server and say, hey, let's just encrypt our stuff. I have no idea who you are, or not really, and you don't have any idea who I am, but let's encrypt it anyway. So this is called opportunistic encryption, and it's of course vulnerable to man in the middle attacks because you're connecting to someone and saying, well, let's encrypt our stuff, but you have no idea who it really is that you're encrypting stuff with. Both might be a good idea anyway. If someone is not actively mounting an attack against you, at least the packets later on will be encrypted and protected from snooping.

So it's a bit of a shame that this protocol that we're looking at here-- SSL, TLS, whatever-- doesn't offer this kind of opportunistic encryption thing. But such is life. So I guess the server [INAUDIBLE] in this protocol. The client sometimes can and sometimes doesn't have to. Make sense? Yeah.

AUDIENCE: I'm just curious. What's to stop someone from-- I mean, let's just say that once a year, they create using new name key pairs. So why couldn't you kind of try to spend that entire year for that specific key?

PROFESSOR: Huh?

AUDIENCE: Why does that not work with this?

PROFESSOR: I think it does work. So OK, so it's like what goes wrong with this scheme. Like, one of the things that we have to do with the topography of good here, and as with Kerberos, people start off using good crypto, but it gets worse and worse over time. Computers get faster. There's better algorithms that are breaking this stuff. And if people are not diligent about increasing their standards, then these problems do creep up. So for example, it used to be the case that many certificates were signed.

Well, there's two things going on. There's a public key signature scheme. And then because the public key crypto has some limitations, you typically-- actually, when you sign a message, you actually take a hash of the message and then you sign the

hash itself because it's hard to sign a gigantic message, but it's easy to sign a compact hash

And one thing that actually went wrong is that people used to use MD5 as a hash function for collapsing the big message here signing into a 128 bit thing that you're going to actually sign with a crypto system. MD5 was good maybe 20 years ago, and then over time, people discovered weaknesses in MD5 that could be exploited.

So actually, at some point, someone did actually ask for a certificate with a particular MD5 hash, and then they carefully figured out another message that hashes to the same MD5 value. And as a result, now you have a signature by a CA on some hash, and then you have a different message, a different key, or a different name that you could convince someone was signed. And this does happen. Like, if you spend a lot of time trying to break a single key, than you will succeed eventually. If that certificate was using crypto, that could be brute force.

Another example of something that's probably not so great now is if you're using RSA. We haven't really talked about RSA, but RSA is one of these public key crypto systems that allows us to either encrypt messages or sign messages. With RSA, these days, it's probably feasible to spend lots of money and break 1,000 bit RSA keys. You'd probably have to spend a fair amount of work, but it's doable, probably within a year easily.

From there, absolutely. You can ask a certificate authority to sign some message, or you can even take someone's existing public key and try to brute force the corresponding secret key, or [? manual hack. ?]

So you have to keep up with the attackers in some sense. You have to use larger keys with RSA. Or maybe you have to use a different crypto scheme.

For example, now people don't use MD5 hashes and certificates. They use SHA-1, but that was good for a while. Now SHA-1 is also weak, and Google is actually now actively trying to push web developers and browser vendors et cetera to discontinue the use of SHA-1 and use a different hash function because it's pretty clear that

maybe in 5 or 10 years time, there will be relatively easy attacks on SHA-1. It's already been shown to be weaker.

So I guess there is no magic bullet, per se. You just have to make sure that you keep evolving with the hackers. Yeah. There is a problem, absolutely. Like, all of this stuff that we're talking about relies on crypto being correct, or sort of being s to break. So you have to pick parameters suitably. At least here, there's an expiration time.

So well, let's pick some parameters that are good for a year as opposed to for 10 years. The CA has a much bigger problem. This key, there's no expiration on it, necessarily.

So that's less clear what's going on. So probably, you would pick really aggressively sort of safe parameters. So 4,000 or 6,000 bit RSA or something. Or another scheme all together. Don't use SHA-1 at all here. Yeah. No real clear answer. You just have to do it.

All right. Any other questions? All right. So let's now look at-- so this is just like the protocol side of things. Let's now look at how do we integrate this into a particular application, namely the web browser?

So I guess if you want to secure network communication, or sort of websites, with cryptography, there's really three things we have to protect in browser. So the first thing we have to protect is data on the network. And this is almost the easy part because well, we're just going to run a protocol very much like what I've been describing so far. We'll encrypt all the messages, sign them, make sure they haven't been tampered with, all this great stuff.

So that's how we're going to protect data. But then there's two other things in a web browser that we really have to worry about. So the first of them is anything that actually runs in the browser. So code that's running in the browser, like JavaScript or important data that's stored in the browser. Maybe your cookies, or local storage, or lots of other stuff that goes on in a modern browser all has to be somehow

protected from network [? of hackers. ?] And we'll see the kinds of things we have to worry about here in a second.

And then the last thing that you might not think about too much but turns out to be a real issue in practice is protecting the user interface. And the reason for this is that ultimately, much of the confidential data that we care about protecting comes from the user. And the user is typing this stuff into some website, and the user probably has multiple websites open on their computer so that the user has to be able to distinguish which site they're actually interacting with at any moment in time.

If they accidentally typed their Amazon password into some web discussion forum, it's going to be disastrous depending on how much you care about your Amazon password, but still. So you really want to have good user interface sort of elements that help the user figure out what are they doing? Am I typing this confidential data into the right website, or what's going to happen to this data when I submit it?

So this turns out to be a pretty important issue for protecting web applications. All right. Make sense?

So let's talk actually what the current web browsers do on this front. So as I mentioned, here for protecting [INAUDIBLE], we're just going to use this protocol called SSL or TLS now that encrypts and authenticates data. It looks very similar to the kind of discussion we've had so far. It includes the certificate authorities, et cetera. And then of course, many more details. Like, TLS is hugely complicated, but it's not particularly interesting from this [INAUDIBLE] angle.

All right, so protecting, [? stopping ?] the browser turns out to be much more interesting. And the reason is that we need to make sure that any code or data delivered over non-encrypted connections can't tamper with code and data that came from an encrypted connection because our threat model is that anything that's unencrypted could potentially be tampered with by a network [? backer. ?]

So we have to make sure that if we have some unencrypted JavaScript code running on our browser, then we should assume that that could've been tampered

with an attacker because it wasn't encrypted. It wasn't authenticated over the network. And consequently, we should prevent it from tampering with any pages that were delivered over an encrypted connection.

So the general plan for this is we're going to introduce a new URL scheme. Let's call HTTPS. So you often see this in URLs, presumably in your own life.

And there's going to be two things that-- well, first of all, the cool thing about introducing a new URL scheme is that now, these URLs are just different from HTTP URLs. So if you have a URL that's HTTPS colon something something, it's a different origin as far as the same origin policy is concerned from regular HTTP URLs. So HTTP URLs go over unencrypted corrections. These things are going over SSL/TLS. So you'll never confuse the two if the same origin policy does its job correctly.

So that's one bit of a puzzle. But then you have to also make sure that you correctly distinguish different encrypted sites from one another. It then turns out cookies have a different policy for historical reasons. So let's first talk about how we're going to distinguish different encrypted sites from one another.

So the plan for that is that actually, the host name via the URL has to be the name in the certificate. So that's what actually turns out that the certificate authorities are going to sign at the end of the day So we're going to literally sign the host name that shows up in your URL as the name for your web server's public key. So Amazon presumably has a certificate for www.amazon.com. That's the name, and then whatever the public key corresponding to their secret key is.

And this is what the browser's going to look for. So if it gets a certificate-- well, if it tries to connect or get a URL that's https://foo.com, it better be the case that the server presents a certificate for foo.com exactly. Otherwise, we'll say, well, we tried to connect to one guy, but we actually have another guy. That's a different name in the certificate that we connected to. And that'll be a certificate mismatch.

So that's how we are going to distinguish different sites from one another. We're

basically going to get the CAs to help us tell these sites apart, and the CAs are going to promise to issue certificates to only the right entities. So that's on the same margin policy side, how we're going to separate the code apart. And then as it turns out-- well, as you might remember, cookies have a slightly different policy. Like, it's almost the same origin, but not quite.

So cookies have a slightly different plan. So cookies have this secure flag that you can set on a cookie. So the rules are, if a cookie has a secure flag, then it gets sent only to HTTPS requests, or along with HTTPS requests. And if a cookie doesn't have a secure flag, then it applies to both HTTP and HTTPS requests.

Well, it's a little bit complicated, right. It would be cleaner if cookies just said, well, this is a cookie for an HTTPS post, and this is a cookie for HTTP host. And they're just completely different. That would be very clear in terms of isolating secure sites from insecure sites. Unfortunately, for historical reasons, cookies have this weird sort of interaction.

So if a cookie is marked secure, then it only applies to HTTPS sites. Well, there's a host also as well, right. So secure cookies apply only to HTTPS host URLs, and insecure cookies apply to both. So that will be some source of problems for us in a second. Make sense? All right.

And the final bit that web browsers do to try to help us along in this plan is for the UI aspect, they're going to introduce some kind of a lock icon that users are supposed to see. So there's a lock icon in your browser, plus you're supposed to look at the URL to figure out which site you're on. Now that's how web browser developers expect you to think of the world. Like, if you're ever entering confidential stuff into some website, then you should look at the URL, make sure that's the actual host name that you want to be talking to, and then look for some sort of a lock icon, and then you should assume things are good to go. So that's the UI aspect of it.

It's not great. It turns out that many phishing sites will just include an image of a lock icon in the site itself and have a different URL. And if you don't know exactly what to look for or what's going on, a user might be fooled by this.

So this UI side is a little messy, partly because users are messy, like humans. And it's really hard to tell what's the right thing to do here. So we'll focus mostly on this aspect of it, which is much easier to discuss precisely. Make sense? Any questions about this stuff so far? Yeah.

AUDIENCE: I noticed some websites that our HTTPS [INAUDIBLE].

PROFESSOR: Yeah. So it turns out that the browsers evolve over time what it means to get a lock icon. So one thing that some browsers do is they give you a lock icon only if all of the content or the resources within your page were also served over HTTPS. So this is one of the problems that forced HTTPS tries to address is this mixed content or insecure embedding kinds of problems.

So sometimes, you will be fail to get a lock icon because of that check. Other times, maybe your certificate isn't quite good enough. So for example, Chrome will not give you a lock icon if it thinks your certificate uses weak cryptography.

But also, it varies with browsers. So maybe Chrome will not give you a lock icon, but Firefox will. So it's, again, there's no clear spec on what this lock icon means. Just people sweep stuff under this lock icon. Other questions?

All right. So let's look at h guess what kinds of problems we run into with this plan. So one thing I guess we should maybe first talk about is, OK, so in regular HTTP, we used to rely on DNS to give us the correct IP address on the server. So how much do we have to trust DNS for these HTTPS URLs? Are DNS servers trusted, or are these DNS mappings important for us anymore? Yeah.

AUDIENCE: They are because the certificate is signing the domain name. I don't think you sign an IP address [INAUDIBLE].

PROFESSOR: That's right. Yeah. So the certificate signs the domain name. So this is like amazon.com. So [INAUDIBLE].

AUDIENCE: Say someone steals amazon.com's private key and [INAUDIBLE] another server with another IP address, and combines [INAUDIBLE] IP address [INAUDIBLE]. But

then you already stole the private key.

PROFESSOR: That's right. Yeah. So in fact, you're describing after both steal the private key and redirect DNS to yourself. So is DNS in itself sensitive enough for us to care about? I guess in some sense you're right, that we need DNS to find the idea, or otherwise we'd be lost because this is just the host name, and we still need to find IP address to talk to it.

What if someone compromised the DNS server and points us at a different IP address? Is it going to be bad? Yeah.

AUDIENCE: Well, maybe just [INAUDIBLE] HTTPS.

PROFESSOR: So potentially worrisome, right. So they might just refuse the connection altogether.

AUDIENCE: Well, no. They just redirect you to the HTTP URL.

PROFESSOR: Well, so certainly, if you connect to it over HTTPS, then they can't redirect. But yeah. Yeah.

AUDIENCE: You can [INAUDIBLE] and try to fool the user. That's [INAUDIBLE].

PROFESSOR: That's right, yeah. So the thing that you mentioned is that you could try to serve up a different certificate. So maybe you-- well, one possibility is you somehow compromised the CA, in which case all right, you're in business. Another possibility is maybe you'll just sign the certificate by yourself. Or maybe you have some old certificate for this guy that you gotten the private key for.

And it turns out that web browsers, as this sort of forced HTTPS paper we're reading touched on, most web browsers actually ask the user if something doesn't look right with the certificate, which seems like a fairly strange thing to do because here's the rule. The host name has to match the name of the certificate, and it has to be valid. It has to be unexpired, all these very clear rules.

But because of historically the way HTTPS has been deployed, it's often been the case that web server operators mis-configure HTTPS. So maybe they just forget to

renew their certificate. Things were going along great and you didn't notice that your certificate was expired and you just forgot to renew it.

So it seems to web browser developers, that seems like a bit of a bummer. Oh, man. It's just expired. Let's just let the user continue. So they offer a dialogue box for the user saying, well, I got a certificate, but it doesn't look right in some way. [INAUDIBLE] go ahead anyway and continue.

So web browsers will allow users to sort of override this decision on things like expiration of certificates. Also for host names, it might be the case that your website has many name. Like for Amazon, you might connect to amazon.com, or maybe www.amazon.com, or maybe other host names. And if you are not careful with the website operator, you might not know to get certificates for every possible name that your website has.

And then a user is sort of stuck saying, well, the host name doesn't look quite right, but maybe let's go anyway. So this is the reason why web browsers allow users to accept more broadly, or a broader range of certificates, than these rules might otherwise dictate. So that's [INAUDIBLE] problem.

And then if you hijack DNS, then you might be able to redirect the user to one of these sites that serves up a incorrect certificate. And if the user isn't careful, they're going to potentially approve the browser accepting your certificate, and then you're in trouble then.

That's a bit of a gray area with respect to how much you should really trust DNS. So you certainly don't want to give arbitrary users control of your DNS name [INAUDIBLE]. But certainly, the goal of SSL/TLS and HTTPS, all this stuff, is to hopefully not trust DNS at all. If everything works here correctly, then DNS shouldn't be trusted.

You can [INAUDIBLE]. You should never be able to intercept any data or corrupt data, et cetera. Make sense? That's if everything works, of course. It's a little bit messier than that.

All right. So I guess one interesting question to talk about is I guess how bad could an attack be if the user mis-approves a certificate? So as we were saying, if the user accepts a certificate for the wrong host or accepts an expired certificate, what could go wrong? How much should we worry about this mistake from the user? Yeah.

AUDIENCE: Well, [INAUDIBLE]. But it could be, [? in example ?], not the site the user wants to visit. So they could do things like pretend to be the user's name.

PROFESSOR: Right. So certainly, the user might then I guess be fooled into thinking, oh, I have all this money, or you have no money at all because the result page comes back saying here's your balance. So maybe the user will assume something about what that bank has or doesn't have based on the result. Well, it still seems bad, but not necessarily so disastrous. Yeah.

AUDIENCE: I think that an [INAUDIBLE] get all the user's cookies and [INAUDIBLE].

PROFESSOR: Right. So this is your fear, yeah. This is much more worrisome, actually, or has a much more longer lasting impact on you. And the reason this works out is because the browser, when it figures out [INAUDIBLE] makes a decision as to who is allowed to get a particular set of cookies or not just looks at the host name in the URL that you were supposed to be connected to.

So if you connect to some attackers' web server, and then you just accept their certificate for amazon.com as the real thing, then the browser will think, yeah, the entity I'm talking to is amazon.com, so I will treat them as I would a normal amazon.com server, which means that they should get access to all the cookies that you have for that host. And presumably they could run a JavaScript code in your browser in that same origin principle.

So if you have another site open that was connecting to the real website-- like maybe you had a tab open in your browser. You closed your laptop, then you opened it on a different network, all of a sudden, someone intercepted your connection to amazon.com and injected their own response. If you approve it, then

they'll be able to access the old amazon.com page you have open because as far as the browser is concerned, these are the same origin because they have the same host name. That's going to be troublesome.

So this is potentially quite a unfortunate attack if the user makes the wrong choice on approving that certificate. Make sense? Any questions about that? All right.

So that's one sort of, I guess, issue that this forced HTTPS paper is worried about is users making a mistake in the decision, users having too much leeway in accepting certificates. Another problem that shows up in practice is that-- we sort of briefly talked about this-- but this is one of the things that also forced HTTPS, I think, is somewhat concerned about is this notion of insecure embedding, or mixed content. And the problem that this term refers to is that a secure site, or any website for that matter, can embed other pieces of content into a web page.

So if you have some sort of a site, foo.com/index.html, this site might be served from HTTPS, but inside of this HTML page, you could have many tags that instruct the browser to go and fetch other stuff as part of this page. So the easiest thing to sort of think about is probably script tags where you can say script source equals http jquery.com.

So this is a popular JavaScript library that makes it easier to interact with lots of stuff in your browser. But many web developers just reference a URL on another site like this. So we should be fairly straightforward, but what's the problem with this kind of set up? Suppose you have a secure site and you just load jQuery. Yeah.

AUDIENCE: It could be fake jQuery.

PROFESSOR: Yeah. So there are actually two ways that you could get the wrong thing that you're not expecting. One possibility is that jQuery itself is compromised. So that seems like, well, you get what you asked for. You asked for this site from jquery.com and that's what you get. If jQuery is compromised, that's too bad. Another problem is that this request is going to be sent without any encryption or authentication over the network.

So if an adversary is in control over your network connection, then they could intercept this request and serve back some other JavaScript code in response. Now, this JavaScript code is going to run as part of this page. And now, because it's running in this HTTPS foo.com domain, it has access to your secure cookies for foo.com and any other stuff you have in that page, et cetera. So it seems like a really bad thing. So you should be careful not to. Or a web developer certainly should be careful not to make this kind of a mistake.

So one solution is to ensure that all content embedded in a secure page is also secure. So this seems like a good guideline for many web developers to follow. So maybe you should just do https colon jquery.com. Or it turns out that URLs support these origin relative URLs, which means you could omit the HTTPS part and just say, [INAUDIBLE] script source equals //jquery.com/ something.

And what this means is to use whatever URL scheme your own URL came from. So this tag will translate to https jquery.com if it's on an HTTPS page, and to regular http jquery.com if it's on a non-HTTPS, just regular HTTP URL. So that's one way to avoid this problem.

Another thing that actually recently got introduced. So this field is somewhat active. People are trying to make things better. One alternative way of dealing with this problem is perhaps to include a hash or some sort of an [? indicator ?] right here in the tag, because if you know exactly what content you want to load, maybe you don't actually have to load it all over HTTPS. You don't actually care who serves it to you, as long as it matches a particular hash.

So there's actually a new spec out there for being able to specify basically hashes in these kinds of tags. So instead of having to refer to jquery.com with an HTTPS URL, maybe what you could do is just say script source equals jquery.com, maybe even HTTP. But here, you're going to include some sort of a tag attribute, like hash equals here, you're going to put in a-- let's say a shell one hash or a shell two hash of the content that you're expecting to get back from the server.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Question?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Ah, man. There's some complicated name for it. I have the URL, actually, in the lecture notes, so [INAUDIBLE]. Subresource integrity or something like this. I can actually slowly be-- well, hopefully will be deployed probably soon in various browsers. Feels like another way to actually authenticate content without relying on data, or data encryption of the [INAUDIBLE].

So here, we have this very generic plan using SSL and TLS to authenticate connections to particular servers. This is almost like an alternative way of thinking of sort of securing your network communication. If the thing you just care about is integrity, then maybe you don't need a secure, encrypted channel over the network. All you need is to specify exactly what you want at the end of the day. Yeah.

AUDIENCE: So doesn't this [INAUDIBLE]?

PROFESSOR: Doesn't this code sit at the client? Well, it runs at the client, but the client fetches this code from some server.

AUDIENCE: [INAUDIBLE]. Can't anybody just [INAUDIBLE]?

PROFESSOR: Yeah. So I think the point of the hash is to protect the containing page from attackers that injected different JavaScript code here. So for jQuery, this makes a lot of sense because jQuery is well known. You're not trying to hide what jQuery source code is. Well, what you do want to make sure is that the network attacker cannot intercept your connection and supply a malicious version of jQuery that's going to leak your cookies.

AUDIENCE: [? Oh, ?] OK.

PROFESSOR: That make sense? It's absolutely true that anyone can compute the hash of these things for themselves. So this is a solution for integrity problems, not for confidentiality. All right.

So this is sort of what I guess developers have to watch out for when writing pages, or including content in their HTML pages on a HTTPS URL. Another worrisome problem is dealing with cookies. And here's where this difference between secure flags and just origins comes into play.

So one thing, of course, the developer could screw up is maybe they just forget to set the secure flag on a cookie in the first place. This happens. Maybe you're thinking my users only ever go to the HTTPS URL.

My cookies are never [INAUDIBLE]. Why should I set the secure flag on the cookie? And they might [? also have the ?] secure flag, or maybe they just forget about it.

Is this a problem? What if your users are super diligent? They always visit the HTTPS URL, and you don't have any problems like this. Do you still leave the secure flag on your cookies? [INAUDIBLE]. Yeah.

AUDIENCE: Could the attacker connect to your URL and redirect you to a [INAUDIBLE]?

PROFESSOR: Yeah. So even if the user doesn't explicitly, manually go to some plain text URL, the attacker could give you a link, or maybe ask you to load an image from a non-HTTPS URL. And then non-secure cookie is just going to be sent along with the network request. So that seems like a bit of a problem. So you really do need the secure flag, even if your users and your application is super careful.

AUDIENCE: But I'm assuming there's an HTTP URL [INAUDIBLE].

PROFESSOR: That's right, yeah. So again, so how could this [? break? ?] Suppose I have a site. It doesn't even listen on port 80. There's no way to connect to me on port 80, so why is it a problem if I have a non-secure cookie?

AUDIENCE: Because the browser wouldn't have cookies for another domain.

PROFESSOR: That's right. So the browser wouldn't send your cookie to a different domain, but yet it still seems worrisome that an attacker might load a URL. So suppose that amazon.com only ever served stuff over SSL. It's not even listening on port 80.

There's no way to connect it.

So in this case, and as a result, they don't set their secure flag on a cookie. So how could a hacker then steal their cookie if Amazon isn't even listening at port 80? Yeah.

AUDIENCE: Can't the browser still think it's an HTTP connection?

PROFESSOR: Well, so if you connect to port 443 and you speak SSL or GLS, then it's always going to be encrypted. So that's not a problem. Yeah.

AUDIENCE: The attacker can [INAUDIBLE] their network.

PROFESSOR: Yeah. So the attacker can actually intercept your packets that are trying to connect to Amazon on port 80 and then appear, and make it appear, like you've connected successfully. So if the attacker has control over your network, they could redirect your packets trying to get to Amazon to their own machine on port 80. They're going to accept the connection, and the client isn't going to be able to know the difference. It will be as if Amazon is listening on port 80, and then your cookies will be sent to this adversary's web server.

AUDIENCE: Because the client is unknown.

PROFESSOR: That's right. Yeah, so for HTTP, there's no way to authenticate the host you're connected to. This is exactly what's going on. HTTP has no authentication, and as a result, you have to prevent the cookies from being sent over HTTP in the first place because you have no idea who that HTTP connection is going to go to if you're assuming a network adversary.

AUDIENCE: So you need network control to do this.

PROFESSOR: Well, yeah. So either you have full control over your network so you know that adversaries aren't going to be able to intercept your packets. But even then, it's actually not so great. Like look at the TCP lecture. You can do all kinds of sequence number of attacks and so on. [? That's going to be ?] troublesome.

All right. Any more questions about that? Yeah.

AUDIENCE: I'm sorry, but isn't the attack intercepted in that case? Is there like a redirect?

PROFESSOR: Well, what that hacker presumably would intercept is an HTTP request from the client going to http amazon.com, and that request includes all your amazon.com cookies, or cookies for whatever domain it is that you're sending your request to. So if you don't mark those cookies as secure, there will be set of both encrypted and unencrypted connections.

AUDIENCE: So how does that request get initiated?

PROFESSOR: Ah, OK. Yeah. So maybe you get the user to visit newyorktimes.com and you pay for an advertisement that loads an image from http colon amazon.com. And there's nothing preventing you from saying, please load an image from this URL. But when a browser tries to connect there, it'll send the cookies if the connection succeeds. Question back there.

AUDIENCE: Will it ask for a change [INAUDIBLE]?

PROFESSOR: Yeah. So HTTPS everywhere is an extension that is very similar to forced HTTPS in some ways, and it tries to prevent these kinds of mistakes. So I guess one thing that forced HTTP does is they worry about such mistakes. And when you sort of opted a site into this forced HTTPS plan, one thing that the browser will do for you is prevent any HTTPS connections to that host in the first place.

So there's no way to make this kind of mistakes of not flagging your cookie as secure, or having other sort of kinds of cookie problems as well. Another more subtle problem-- so this, the problem we talked about just now is the developer forgetting to set the secure flag on a cookie. So that seems fixable. OK, maybe the developer should just do it. OK, fix that problem.

The thing that's much more subtle is that when a secure web server gets a cookie back from the client, it actually has no idea whether this cookie was sent through an encrypted connection or a plain text connection because when the server gets a

cookie from the client, all it gets is the key value pair for a cookie. And as we sort of look at here, the plan for the [INAUDIBLE] follows is that it'll include both secure and insecure cookies when it's sending a request to a secure server, because the browser here was just concerned about the confidentiality of cookies.

But on the server side, you now don't have any integrity guarantees. When you get a cookie from a user, it might have been sent over an encrypted connection, but it also might have been sent over a plain text connection. So this leads to somewhat more subtle attacks, but the flavor of these attacks tend to be things like session fixation. What it means is that suppose I want to see what emails you're sending.

Or maybe I'll set a cookie for you that is a copy of my Gmail, cookie. So when you go to compose a message in Gmail, it'll actually be saved in my sent folder inside of your sent folder. It'll be as if you're using my account, and then I'll be able to extract things from there.

So if I can force a session cookie into your browser and sort of get you to use my account, maybe I can extract some information that way from the victim. So that's another problem that arises because of this grey area [INAUDIBLE] incomplete separation between HTTP and HTTPS cookies. Question.

AUDIENCE: So you would need a [INAUDIBLE] vulnerability to set that cookie [INAUDIBLE].

PROFESSOR: No. [INAUDIBLE] vulnerability to set this cookie. You would just trick the browser into connecting to a regular HTTP host URL. And without some extension like forced HTTPS or HTTPS everywhere, you could then, as an adversary, set up a key in the user's browser. It's a non-secure cookie, but it's going to be sent back, even on secure requests.

AUDIENCE: So do you have to trick the browser into thinking the domain is the same domain?

PROFESSOR: That's right. Yeah. So you have to intercept their network connection and probably do the same kind of attack you were talking about just a couple of minutes ago. Yeah. Make sense?

All right. So I guess there's probably [INAUDIBLE]. So what does forced HTTPS actually do for us now? It tries to prevent some subset of these problems. So I guess I should say, so forced HTTPS, the paper we read was sort of a research proposal that was published I guess five or six years ago now. Since then, it's actually been standardized and actually adopted.

So this was like a somewhat sketchy plug-in that stored stuff and some cookies. Are they worried about getting evicted and so on? Now actually, most browsers look at this paper and say, OK, this is a great idea.

We'll actually implement it better within the browser itself. So there's something called HTTP strict transport security that implements most of the ideas from forced HTTPS and actually make a good story. Like, here's how research actually makes an impact on I guess security of web applications and browsers.

But anyway, let's look at what forced HTTPS does for a website. So forced HTTPS allows a website to set this bit for a particular host name. And the way that forced HTTPS changes the behavior of the browser is threefold.

So if some website sets forced HTTPS, then there's sort of three things that happen differently. So any certificate errors are always fatal. So the user doesn't have a chance of accepting incorrect certificate that has a wrong host name, or an expiration time that's passed, et cetera.

So it's one thing that the browser now changes. Another is that it redirects all HTTP requests to HTTPS. So this is a pretty good idea. If you know a site is always using HTTPS legitimately, then you should probably prohibit any regular HTTP requests [? website ?], because that's probably a sign of some mistake or attacker trying to trick you into connecting to a site without encryption. You want to make sure this actually happens before you issue the HTTP request. Otherwise, the HTTP request has already sort of sailed onto the network.

And the last thing that this forced HTTPS setting changes is that it actually prohibits this insecure embedding plan that we looked at below here when you're including a

HTTP URL in an HTTPS site. Make sense? So this is what the forced HTTPS sort of extension did. In terms of what's going on now is that well, so this HTTPS strict transport security HSTS protocol basically does the same things.

Most browsers now prohibit insecure embedding by default. So this used to be a little controversial because many developers have trouble with this. But I think Firefox and Chrome and IE all now by default will refuse to load insecure components, or at least secure JavaScript and CSS, into our page unless you do something. Question.

AUDIENCE: Don't they prompt the user?

PROFESSOR: They used to, and the user would just say, yes. So IE, for example, used to pop up this dialogue box, and this paper talks about, saying, would you like to load some extra content, or something like that.

AUDIENCE: [INAUDIBLE] because [INAUDIBLE].

PROFESSOR: Yeah. I think if you try to pretend to be clever, then you can bypass all these security mechanisms. But don't try to be clever this way. So this is mostly a non-problem in modern browsers, but these two things are still things that forced HTTPS and HTTP strict transport security provide and are useful. Yeah.

AUDIENCE: What happens when a website can't support HTTPS? [INAUDIBLE] change their [INAUDIBLE]?

PROFESSOR: So what do you mean can't support HTTPS?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Well, OK. So if you have a website that doesn't support HTTPS but sets this cookie, what happens?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. So this is the reason why it's an option. So if you opted everyone, then you're

exactly in this boat. Like, oh, all of a sudden, you can't talk to most of the web because they don't use HTTPS. So you really wanted this to be selectively enabled for sites that really want this kind of protection. Yeah.

AUDIENCE: But also, if I remember correctly, you can't set the cookie unless the site [INAUDIBLE].

PROFESSOR: That's right, yeah. So these guys are also worried about denial of service attacks, where this plug in could be used to cause trouble for other sites. So if you, for example, set this forced HTTPS bit for some unsuspecting website, then all of a sudden, the website stops working because everyone is now trying to connect to them over HTTPS, and they don't support HTTPS. So this is one example of worrying about denial of service attacks.

Another thing is that they actually don't support setting forced HTTPS for an entire domain. So they worried that, for example, at mit.edu, I am a user at mit.edu. Maybe I'll set a forced HTTPS cookie for start.mit.edu in everyone's browsers. And now, only HTTPS things work at MIT. That seems also a little disastrous, so you probably want to avoid that.

On the other hand, actually, HTTPS strict transfer security went back on this and said, well, we'll allow this notion of forcing HTTPS for an entire subdomain because it turns out to be useful because of these insecure cookies being sent along with a request that you can't tell where they were sent from initially. Anyway, so there's all kinds of subtle interactions with teachers at the lowest level, but it's not clear what the right choice is.

OK, so one actually interesting question you might ask is are these fundamental to the system we have, or are these mostly just helping developers avoid mistakes? So suppose you had a developer that's very diligent and doesn't do insecure [INAUDIBLE] embedding, doesn't solve any other problems, always gets their certificates renewed, should they bother with forced HTTPS or not? Yeah.

AUDIENCE: Well, yeah. You still have the problem with someone forcing HTTP protocol. Nothing

stops the hacker from doing [? excessive ?] [INAUDIBLE] forces the user to load something over HTTP and then to intercept the connection.

PROFESSOR: That's true, but if you feel they're very diligent and all their cookies are marked secure, then having someone visit an HTTP version of your site, shouldn't be a problem.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. So you'd probably have to defend against cookie overwrite or injection attacks, and that's sort of doable. It's a little tedious, but you can probably do something.

AUDIENCE: Yeah. I think her point is that also, it didn't-- security didn't check the certificate, right?

PROFESSOR: Yeah. So that's one. I think that this is the biggest thing is this first point, which is that everything else, you can sort of defend it against by cleverly coding or being careful in your application. The first thing is something that the user has-- or the developer-- has no control over because the developer wants to make sure, for example, that their cookie will only be sent to their server as signed by this CA.

And if the user is allowed to randomly say, oh, that's good enough, then the developer has no clue where their cookie's going to end up because some user is going to leak it to some incorrect server. So this is, I think, the main benefit of this protocol. Question back there.

AUDIENCE: [INAUDIBLE] second point is also vital because the user might not [INAUDIBLE]. You might [INAUDIBLE] of the site, which would be right in the middle.

PROFESSOR: I see. OK. So I agree in the sense that this is very useful from the point of view of UI security because as far as the cookies are concerned, the developer can probably be clever enough to do something sensible. But the user might not be diligently looking at that lock icon and URL at all times.

So if you load up amazon.com and it asks you for a credit card number, you might

just type it in. You just forgot to look for a lock icon, whereas if you set forced HTTPS for amazon.com, then there's just not chance that you'll have an HTTP URL for that site. It still [? causes a ?] problem that maybe the user doesn't read the URL correctly. Like it says Ammazon with two Ms dot com. Probably still fool many users.

But anyway, that is another advantage for forced HTTPS. Make sense? Other questions about this scheme? All right.

So I guess one interesting thing is how do you get this forced HTTPS bit for a site in the first place? Could you have intercepted that as an attacker and prevent that bit from being set if you [? want to mount a fax? ?] Yeah.

AUDIENCE: [INAUDIBLE] HTTPS. I mean, HTTPS, we're [? assuming ?] [INAUDIBLE] protocol [INAUDIBLE].

PROFESSOR: That's right. So on one hand, this could be good. But this forced https that can only be sent over HTTPS connection to the host in question. On other hand, the user might be fooled at that point.

Like, he doesn't have the forced HTTPS bit yet. So maybe the user will allow some incorrect certificate, or will not even know that this is HTTP and not HTTPS. So it seems potentially possible for an attacker to prevent that forced HTTPS bit from being sent in the first place. If you've never been to a site and you try to visit that site, you might never learn whether it should be forced HTTPS or not in the first place. Yeah.

AUDIENCE: Will the [INAUDIBLE] roaming certificate there.

PROFESSOR: That's right, yeah. So I guess the way to think of it is if they did a set, then you know you talked to the right server at some point, and then you could continue using that bit correctly. On the other hand, if you don't have that bit set, or maybe if you've never talked to a server yet, there's no clear cut protocol that will always give you whether that forced HTTPS bit should be set or not.

Maybe amazon.com always wants to set that forced HTTPS bit. But the first time

you pulled up your laptop, you were already on an attacker's network, and there's just no way for you to connect to amazon.com. Everything is intercepted, or something like this. So it's a very hard problem to solve. The bootstrapping of these security settings is pretty tricky.

I guess one thing you could try to do is maybe embed this bit in DNSSEC. So if you have DNSSEC, already in use, then maybe you could sign whether you should use HTTPS or not, or forced HTTPS or not, as part of your DNS name. But again, it just boils down the problem to DNSSEC being secure. So there's always this sort of rule of trust where you have to really assume that's correct. Question.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. So I guess Google keeps trying to improve things by hard coding it. So one thing that Chrome offers is that actually, the browser ships with a list of sites that should have forced HTTPS enabled-- or now, well, this HSTS thing, which is [INAUDIBLE] enabled. So when you actually download Chrome, you get lots of actually useful stuff, like somewhat up to date CRL and a list of forced HTTPS sites that are particularly important.

So this is like somewhat admitting defeat. Like the protocol doesn't work. We just have to distribute this a priori to everyone. And it sets up this unfortunate dichotomy between sites that are sort of important enough for Google to ship with the browser, and sites that don't do this.

Now of course, Google right now tells you that anyone can get their site included because the list is so small. But if this grows to millions of entries, I'm sure Google will stop including everyone's site in there. But yeah, you could totally add a domain. And you could email Chrome developers and get your thing included on the list of forced HTTPS URLs.

Anyway, any other questions about forced HTTPS and SSL? All right. Good. So I'll see you guys on Wednesday at the [INAUDIBLE].