We started last time on protection.

You remember the model that we're considering?

We've got some client communicating over the Internet with some server.

And we said that there are three main technical problems that we're going to look to address.

There is the problem of authentication, the problem of authorization and the problem of confidentiality.

Today what we're going to do is we're going to talk mostly about this issue of authentication, but I want to just return to some of the details that we were talking about at the end of class last time so that you guys all get the important bits out of our discussion of the RSA protocol.

Remember, last time we started talking about these are the main requirements of a protection system.

And we drew a diagram that said that at the top level we want to have an application that is protected.

Underneath that we're going to have a set of security primitives.

And then, underlying all of this, we're going to have cryptography.

Cryptography is going to be sort of the essential technique that we use to enable protection.

We talked briefly about several different types of cryptography.

Remember, we had our model where we have A is sending through some box which takes the data and transforms it using some cryptographic transform.

And then that cryptographically transformed data gets transmitted and then is reverse transformed or untransformed at the other side and it comes out to the receiver.

What cryptography does is we said that we want this open model where there is some key k1 here and some key k2 here that are used to decide how this transform operator works and this untransform operator works on the other side.

We talked about a shared key case where k1 is equal to k2. And the appendix A of the chapter in the notes goes over a shared key protocol called DES that is fairly commonly used.

The basic idea behind shared key cryptography is typically that both sides have this key and we're going to transform the data by combining the message stream that is coming from A together with the key in some way, say, for example, by xoring the data in many different ways or permuting the data around and then xoring it with the key in some combination of ways.

That's sort of what the DES protocol does.

It's just multiple rounds, basically, of permuting and xoring the data over and over again.

The idea is that if an attacker doesn't know the key, it's very hard for them to reverse this process without having the key, but if you know the key it's relatively easy to reverse the process.

We also talked a little bit about public key cryptography where k1 is not equal to k2. In this case, we started talking about this protocol RSA which is an example of a public key protocol.

If you remember, RSA has some set of rules for actually doing the transformation of data.

The basic idea is that we're going to get a private key and a public key here, which is going to consist of this prime number plus the product of these two prime numbers and this private key k which is going to consist of the other prime number and the product of these two things.

Now we have this way in which we can transform the data.

What this transform box does is simply raise the message to the power of e modulo n and we have the reverse transform.

This is just a specific mathematical operation.

The reason that this mathematical operation works is that the intuition is that it is relatively easy to take something to an exponent.

I can take a number and exponentiate it without doing very much work.

But in order for an attacker to compromise this system, that is in order to discover, for example, what the private key is, in this case, what this value d is, given just the public key which consists of e and n, the attacker is going to be able to factor this number n, which turns

out to be very difficult to do.

Essentially, breaking this cryptographic system comes down to be able to factor a very large number.

And that turns out to be, computationally, very, very difficult to do.

So that's sort of the intuition behind the protection of this scheme.

The nice thing about RSA is it has, in most public key cryptographic protocols is that they have this property that the reverse is also true.

So I can transform using exponentiation with the public key or the private key and I can reverse transform using the opposite.

If I transform with the public key, I can reverse transform with the private key and vise versa.

So we are going to exploit this property a lot when we're talking about this kind of cryptography.

Just to sort of point out that this is hard, this is an estimate from the RSA Security Company about how many compute years on modern computers it would take to factor an RSA key of a given link.

So the bottom is the key link in bytes on the x-axis and then the y-axis is the machine years to factor.

If the machine years to factor was 1,000, that means it would take 1,000 machines working together one year to do this or one machine 1,000 years to do it.

So you see that the y-axis here is going up exponentially.

So to factor a 500 byte key is not that hard.

It takes maybe a few months of compute time to do it.

To factor a 2,000 byte key takes this a ridiculously long time, ten to the 16th years it is going to take to factor this thing.

So there is just no way that, at least modern computers, are ever going to be able to actually

perform this factoring using currently known algorithms for factoring.

It is, of course, possible that somebody will develop a very fast factoring algorithm or that there will be some fundamental breakthrough like quantum computing that will make it possible to factor very large numbers.

But, at this point, this is sort of the estimate of how long it would take to do this.

So, by using long keys, you can be sort of guaranteed that somebody won't be able to actually factor this number.

So by using a factoring-based attack on RSA that sort of won't be able to break it in this way.

The largest RSA number that has been factored to date is some 500 odd byes, something like 520 bytes, so RSA has challenges for the largest numbers that can be factored.

So we're sort of just barely on this curve is where the largest numbers that have been factored to date.

And these are typically some collection of machines on the Internet.

Somebody gets 1,000 machines on the Internet and they chew on this problem.

I think, in this case, it took 1,000 machines three months to find a factor of this 500 byte key.

This is the basic idea now.

What we're going to do is use these cryptography algorithms, this math to build up a set of primitives that we can use to actually accomplish our security goals that we want.

And I put these on the board briefly last time.

But I didn't talk at all about what the actual functionality is.

And we're going to spend a lot of time talking about two of these primitives today, and then we'll talk about a few more of them next time.

We have two sets of primitives in particular that we are interested in.

One set is called sign and verify.

And the other set is called encrypt or decrypt.

And we're going to use sign and verify basically as a way to achieve authentications.

Remember, authentication is the process of verifying that a message actually came from who it was supposed to come from.

And we're going to use encrypt and decrypt in order to provide confidentiality to guaranty that somebody who we don't want to be able to read a message, somebody who shouldn't be able to read a message can, in fact, not read that message.

Let me just illustrate a simple example.

Suppose we're using a public key system, we can say suppose we use something like RSA to encode a message m with some key, say we're sending a message from A to B, we use it to encode some key, say, kA private.

What this notation means is that whoever has A's private key encodes that message or transforms that message using a cryptographic protocol using, say, for example, kA private.

What this says is Alice, perhaps as A, runs her RSA algorithm on the message and generates some encoded thing.

This is an encryption.

We're going to use this encoding with the private key to sign a message.

When Alice encodes this with her private key what that means is that somebody who has Alice's public key can decode this, because we said if you encode with the private key somebody with the public key can decode.

For example, if she sends this to Bob, Bob is going to be able to decode this.

At first that seems like, well, what good does that do us?

Bob can read this message.

But the thing is if Bob can decode this, what he knows, with very high certainty is that the person who encoded this message had access to A's private key.

If he is willing to trust that, in fact, the only person who has access to A's private key is A

herself, then he can be reasonably confident that this message came from A.

You guys all sort of see how that works.

What we say is that Alice signs this message.

And, by signing this message, she is able to sort of give Bob some confidence that this message, in fact, came from her.

We can also do the opposite thing where, say, for example, Alice is sending a message to Bob.

She can take m and sign it with kB public.

Sorry, she can encrypt it with kB public.

She is going to use the RSA algorithm.

She is going to encrypt this message.

And, when she encrypts this message, she is going to basically use the RSA algorithm using kB public.

And what is going to happen is now she is going to be reasonably assured that this message is going to be able to only be read by Bob because only Bob presumably has his own private key.

What I've shown here makes this look very simple.

I sort of said it as though we're simply taking this message and running the basic RSA algorithm, as I showed it, on the message in order to do sign and in order to encrypt.

It turns out that actually getting sign and encrypt to work property requires some fairly sophisticated sort of reasoning about the RSA algorithm.

For example, when you're encrypting something, if the message is very small, it turns out that if you encrypt a small message simply using the RSA algorithm as I presented it that it's relatively easy to break the RSA algorithm.

The RSA algorithm becomes vulnerable with very small messages.

That means the encryption process actually needs to pad out the message to some longer link.

It needs to put unused bytes at the end of every message in order for it to be secure.

These sign and encrypt algorithms themselves sort of have to do some additional work on top of the basic RSA algorithm in order to be secure.

And that additional work is sort of beyond the scope of this class.

It requires a certain degree of mathematical sophistication that we are not going to rely on here.

You can go take an extra class about cryptography and learn about how these sign and encrypt algorithms actually work, but the basic idea is that you can sort of grasp it by just seeing RSA.

These things are hard to build.

Now that we have sort of quickly reviewed the basics of protection and we've seen the sign and verify and encrypt and decrypt primitives, now what I want to do is to talk a little bit about how we can actually start building up, solving this problem of authenticating a user.

When we authenticate a user we have two objectives.

One thing we're trying to achieve is to determine who is making a request.

In practice, it's going to turn out, in fact, that we may not officially be able to say exactly the specific person who is being able to make the request.

We may only be able to say that the same principle or person or computer as before is making the request, or the same principle who had some special privileged piece of information that the person before had is making this request.

We may not be able to actually associate this with a physical person, but at least we're going to be able to know that we've seen somebody who had this same information before.

We're trying to make sure that this is sort of consistent with a series of requests that we've seen in the past.

A simple example is I lot onto Amazon, I create a password.

Now, anybody who I give that password to can log onto Amazon.

And all the system can say is that it knows that whoever it is who is accessing definitely has my password information and sort of it doesn't know officially that it is actually the same physical person.

The other thing, though, that we want authentication to do is to tell us basically that the message that was sent is equal to the message that was received.

We want it to be the case that if Alice sends a message to Bob, that when Bob receives that message, we want him to have some assurance that, in fact, that message was actually the message that Alice tried to send.

And we're going to talk about how this is sort of going to relate to the problem of integrity that we talked about before, verifying that a message hasn't been corrupted in transmission.

But this is actually a harder requirement because we might need to not only worry about sort of random bits being flipped by the communication channel but we need to worry about some malicious person coming in and physically actively modifying the bytes of the message.

Authentication is sort of the first step that we need to have in any secure system because authentication is going to be the key.

We have to authenticate a user before we can authorize the user.

Before we can say, yes, this user is allowed, we have to figure out who the user is.

And, similarly, we can use authentication, this process of figuring out who a user is to help us to build up audit trails.

Remember we said audit trails are something that is important.

Once we figure out who a user is, we can keep logs of which user it was that connected to our machine.

Let's look at a simple model of how authentication works.

The idea is as follows.

We have our principle, say, for example, a user who is sending a request to some server.

And this request might be something, for example, like buy Apple stock.

And the server's job is to determine whether this request is actually authentic, whether it actually came from the user it claims to come from and whether the message that was received is actually the message that was sent.

The way it's going to do this is we're going to introduce something we call a guard that is at the front end of this server that takes in all the incoming requests and is in charge of authenticating them.

And then this guard is going to dispatch results off to any of the services that the server provides.

It is going to say if the request is authentic it is going to invoke the service that we need.

In other words, this guard is the type of mediator that sits in between all the requests between the principle and the service that it is trying to access and the guard is going to be the thing that does all of our authentication.

Now the question is what is it exactly that the guard does?

What are the processes that the guard has to go through in order to authenticate a user?

And, before we talk about the specific technical solution, I want to sort about two things that are really going on when you're trying to figure out who it is that is requesting and whether you're going to allow that person to do what they want to do.

So the issue is, essentially, as follows.

There is this question about is the computer that is making this request actually have the appropriate keys to make this request?

Does it have the appropriate password to be allowed to make this request?

Then there is this other question which is do I trust this user?

Is this somebody who I want using my service?

Suppose that I go to E*Trade and say I want to buy Apple.

How does E*Trade decide whether or not it trusts me, whether I'm somebody who is allowed to buy Apple stock from them or not?

So they have some process for doing that.

You have this mechanical process of authentication.

And then we have this sort of psychological process of determining whether or not you trust somebody.

So this authentication is going to be a technical thing and we are going to describe this as being psychological.

What these things are connected by is some notion of a name.

What the authentication says is, for example, Sam wants to buy Apple.

It determines that I am, in fact, the person who is making the request, that it is able to verify that I am the one who is actually making the request.

What trust is responsible of doing is determining whether or not Sam should actually be allowed to make this request.

And this is not something that we necessarily have a technical solution to.

Different computer systems have different ways of determining whether or not they should trust each other.

For example, when you go to a service on the Web, you make a decision about whether you trust that service based on some sort of way that you have of evaluating the service.

Maybe your Mom told you that E*Trade is the best place to buy stocks on the Internet with.

You trust your Mom and so, therefore, you trust E*Trade.

Or maybe you saw an advertisement for E*Trade and you think any company that has enough money to advertise on the side of the freeway must be a trustworthy company and, therefore, I trust you.

Similarly, E*Trade, presumably determines that they trust you based on, well, one thing they might do is, you might give them a bank account number, and your bank might say, yes, this person has enough money to make this request.

That is a way that they might decide that they trust you.

Alternatively, E*Trade gives out loans, so maybe they go talk to your credit reporting agency, and the credit reporting agency says, yeah, this guy is trustworthy.

He has a high credit score so, therefore, E*Trade will let you borrow $1,000 to invest in the stock market or something.

This issue of trust is one that we don't have a technical solution to, but it is absolutely an essential part of authenticating and authorizing a user to use a computer system.

So it's worth just bearing this in mind that any time you're building a computer system you need to think about how it is that you determine whether a user is trustworthy or not.

And this goes both ways, both for the user of the system, how do they determine that you're trustworthy, and for the system, how do you determine that the users are trustworthy?

Now the key problem is focusing on this sort of authentication problem.

What is the technical meat of how we're actually going to authenticate a user?

One of these problems is determining that this message that was sent was actually the message that was received.

Here is a simply proposal that I might have which is I might compute one of these CRC checks.

These checksums that we talked about when we were talking about networking that we used to determine whether or not the message has been corrupted in transmission.

It turns out that is not going to work.

You might also think the way that you are going to insure that the person who is requesting the message is actually who they claim to be is, for example, using our encryption primitive.

If we had a one time pad available to us, we might encrypt the entire stream of data that is

going across.

And then we would think well, there is no way that somebody on the other end, unless they are authorized to read this, will actually be able to read it.

And, therefore, this is going to give us some guaranty about who is making the request.

Neither of these things quite works, and let me explain why.

Integrity, in the sense that we studied it previously in 6.033, is not equal to authenticity, is not equal to confidentiality.

And the reason is as follows.

Let's just look at a simple example.

Suppose there is Alice who is sending a message to some server, and suppose this server is something where Alice doesn't really mind if her request is seen in public but she wants to make sure that the server actually receives the request that she made.

So she cares about it being the right request, but she doesn't care about it being a secret request.

Maybe Alice says I want to donate $100 to Save the Whales.

She says I don't care if anybody knows that I want to save the whales.

That's a noble and respectful cause.

In fact, I am proud of the fact that I want to save the whales.

So anybody who wants to can look at this message.

But I really only want to give Save the Whales $100. I want to make sure that somebody doesn't mess with this message and some Save the Whale activist doesn't change this message to be $10,000 so that I don't lose a bunch of money.

Let's look at some techniques that we might use to sort of guaranty that this is the case.

One thing we might do is, for example, use a one time pad.

Suppose Alice and the server share a random bit string.

Alice might encode, might xor the message that she transmits, and then the server might apply the xor again with the same bit string and get the message back.

The problem with this is that this doesn't prevent some malicious user or Lucifer from coming into the middle of this and interfering with this byte string as it is transmitted.

Even though Lucifer cannot actually look at the message, he can manipulate the byes to his heart's content.

And he may get lucky and change the bytes so that it says something like donate some different amount of money to save the whales or he may just garble Alice's requests so it cannot be processed.

So one time pad is not going to work.

Now, the other alternative I said we can talk about would be something like using a CRC.

What Alice would do is send m followed by the CRC of m.

The problem with this is that now Lucifer can look at this message as it comes across the wire and the checksum, and he can simply change the message to say whatever he wants and then recomputed the checksum.

So the checksums, as we have talked about them so far, are good for handling the case of non-malicious attacks or non-malicious problems like a byte gets changed in transmission or there is some error in transmission, but they don't solve this problem of preventing somebody from tampering with the message and being able to detect when that tampering also effects the CRC.

If you think about this for a minute, the property that we want sort of feels like some combination of these two things.

Basically, what we want is some checksum-like operation such that we know that the only person who could have possibly written this checksum was somebody who access to some key that Alice has like Alice's private key.

We want a checksum that is dependent on a key, and we want this checksum to have some

fairly strong properties.

We want this checksum to be resistant to these kinds of attacks that we talked about Lucifer applying.

In particular, we want the checksum to be resistant to Lucifer modifying the message.

So the checksum should detect that this message was modified, if it was modified.

We also want it to be able to detect other transformations of the message.

For example, suppose that rather than modifying m, changing something, Lucifer just reorders some of the words in it, keeps exactly the same characters and exactly the same length but just switches some things around.

And there are other kinds of things that we might want to be able to resist as well.

For example, taking exactly the same message and sticking a little bit of extra data onto the end of it, appending something to the message.

There is a bunch of different sort of properties that we want whatever this thing is that is going to allow us to do authentication to have.

Let's look at the sort of model for how our basic authentication system is going to work.

It is really pretty straightforward and is going to rely on using our sign and verify primitives that I am erasing.

The idea is as follows.

We have Alice over here that is sending a message.

That message is going to go into our sign cryptographic primitive, it's going to be transformed and is going to come over here to our verify box, and then it is going to come out on the other side at the server.

And the sign is going to have some k1 that gets fed into it and verify is going to have some k2 that gets fed into it.

And the idea is that in addition to signing the message we are also going to transmit the message itself.

So this is going to be sort of like this idea of putting a CRC on a message, but this thing that we are going to do when we sign is going to provide this sort of better properties than the basic CRC did.

The idea is this is a small bit of additional information that gets transmitted with the message that allows the server to be confident that, in fact, the message that was transmitted came from Alice and that it has not been corrupted along the way.

This is the basic idea now k1 comes in, it gets transmitted, goes through the verify box, and the verify box reports yes or no, this message was correct or was not correct.

And the server also has access to the message.

If the server gets yes in the message then it continues to process it.

And we have two kinds of ways of doing authentification that correspond to shared key and public key cryptography.

If k1 is equal to k2 we say that this little bit of information that we have transmitted is a MAC, a message authenticator.

When you are using shared key cryptography that is the bit of additional information that gets transmitted.

And when k1 is not equal to k2, that is if we're using public key cryptography, we call this bit of additional information a signature.

To give you a very simple example of what a signature might consist of, suppose we have a message m.

In public key cryptography, a simple kind of signature that you can compute is to take the hash of m.

And then take that thing and encode it with Alice's private key.

This is a simple kind of a signature that we might attach to a message.

And this has the properties we want, which is that we believe that only Alice could have actually encoded this message because we belief that only Alice has access to her private key.

And so when somebody uses the public key to decode this they get something back which is a hash of the message.

And we call this a cryptographically secure hash.

It is some way of combining the bits of the message together such that when the server computes the hash of m, the probability that it matches this hash that was in the signature that Alice sent, they will match if the messages are the same.

And, if the hash was computed on a different message than the message that the server receives, the probability is very, very low that these two messages are, in fact, the same.

So they say it has a low collision probability.

It's very unlikely that some m prime that is not equal to m has the same hash as m.

That's what a cryptographically secure hash provides us.

And, again, cryptographically secure hashes are another kind of sort of mathematical technique.

There is a set of algorithms for driving these things.

The appendix talks about a common one which is used called sha-1. And sha-1, it turns out, has recently been shown to have some higher probability of collisions than was previously thought.

And there is a new protocol called sha-256 which is still believed to be secure.

But sha-1, the S-H-A, which stands for secure hash, is one that is very commonly used in the world today.

So this is a simple example of a signature and it allows basically for the server to verify one by decoding the message with Alice's public key that the message actually came from Alice, and this hash allows it to be reasonably assured that the message has not been tampered with.

If the message had been tampered with the hash wouldn't match, and if this signature had been tampered with then the receiver wouldn't have been able to decode this thing and get something back that made any sense.

Or, again, the hash probably wouldn't match.

This is the basic solution for doing authentication.

This screen is too low.

Now that we've sort of seen a method for authenticating a message, there are sort of a bunch of little details that we left hanging.

In particular, one of the details that we haven't yet addressed is, suppose we are using some public key based mechanism like this, I haven't yet told you or we haven't talked about how it is that somebody would learn Alice's public key?

We sort of just assumed that whoever is receiving the message is able to figure out what Alice's public key is.

And one thing you could imagine is that Alice goes around and physically meets everybody who she would ever want to give a key to and she hands them a slip of paper that has her public key written on it.

But, obviously, one, that's not how the Internet works clearly because we have some cryptographic protocols that allow us, we don't have to meet everybody and we can still get established secure communication channels.

And, two, that just doesn't sound very scalable.

It doesn't sound like a great solution.

This is the key distribution problem.

This is going to apply mostly for the case of public key cryptography.

In the case of shared key cryptography, this method I am going to show you in a minute is not going to apply necessarily to shared key cryptography, but I will show you a way in which we can use public key cryptography to sort of bootstrap or to exchange a shared key which we can then use over our communication channel.

This is the key distribution problem for public key cryptographic systems.

One solution, if Alice cannot physically meet with Bob, one thing that you might imagine doing

is you could just have Alice send a message to Bob saying here is my public key.

Or, better yet, Alice's public key is, A's public key is X.

But this isn't a very good approach because how does B actually know that this was Alice that sent this message?

This could have been some Lucifer who said, oh, by the way, Alice's key is this.

And now B thinks that Alice's key is, in fact, Lucifer's key.

And now Lucifer can decode any message that B wants to send to A.

So this isn't really a very good approach because if A just, sort of out of the blue, blurts out to be my key is this then it doesn't have any way of actually knowing that this was, in fact, A who reported this key.

That doesn't solve our problem.

Instead, we are going to use a technique called certificates.

Certificates work like this.

The idea with certificates is that we are going to introduce a third-party who I have shown here as Charles.

Let's suppose that Alice and Bob know and trust Charles.

So they have already somehow exchanged information with Charles, exchanged keys with him.

The idea is now suppose Alice wants to send a message to Bob.

What she does is she sends this message, and she signs the message using her private key.

That is what this S parentheses m kApriv means.

Now what Bob is going to do is say, well, I don't know who this Alice is.

And rather than simply asking Alice for her public key, which we have already said is not a very good idea, what Bob does is he asks Charles for her public key because he trusts Charles and

he knows that Charles probably knows something about Alice.

And what Charles responds with is Alice's public key, as well as he needs to make sure that he, himself, signs this message with his own private key so that somebody else cannot intercept the message as it is coming back and overwrite it.

He put his signature on this message that he sends back with Alice's public key in it.

Now Bob has Alice's public key, and he can go ahead and decode this message, he can go ahead and verify this message that Alice tried to send to him.

So, in this, we call this Charles here, sometimes it is called a CA, a certificate authority.

And the idea is that these CAs, a small number of CAs can service a very large number of users.

So rather than having to have every user exchange keys with every other user, each user just exchanges keys with a few certificate authorities.

And then, those certificate authorities, sort of act like these hubs that propagate keys out into the rest of the users.

This is a very simple way in which you might be able to disseminate a public key.

We will talk a little bit later on about sort of a more formal way of reasoning about how public keys are disseminated and how these kinds of webs of trust are built up, but this is just a simple way to sort of start thinking about how public keys might be disseminated automatically over the Internet.

What I want to do now is just briefly turn, with the last few minutes, to this question of how we actually establish a secure communication channel between two parties.

This is going to get at our property, the sort of establishing a confidential communication channel.

We are going to talk a little bit about how we establish confidentiality.

And so the idea is that we want to find some way that we can encrypt the communication between our two parties.

And what we are going to do is first authenticate the user using a technique like this, and then what we are going to do is use public key to authenticate.

This is establishing a secure communication channel.

We are going to use public key to authenticate.

But what often happens in practice in the Internet is that you use public key to authenticate, and then what you are going to do is use a shared key cryptography protocol to actually encrypt the information that is flowing back and forth between the two parties.

And the reason for that is if you sort of looked at, even though I said that these public key methods like RSA, it's easier to exponentiate than it is to factor.

It turns out that exponentiation in RSA is still relatively expensive because exponentiation requires, these keys are very long, they are thousands of bytes long, so you are taking these long messages and taking them to very large powers.

Doing that is computationally expensive.

You get these very big numbers that computers are not terribly good at handling.

And so it has a big impact on performance if you use public key cryptography all the time.

Instead, often times what is done is we use public key cryptography to sort of, as I said, bootstrap the process of exchanging a shared key between two servers.

That is what we are going to see.

We are going to see how we can then exchange a shared key which we can then use to encrypt the communication between two parties.

This is a little puzzle for you guys.

What I am going to put up now is a broken protocol and see if you can figure out, in the next few minutes, how it is broken.

It is not very broken but it is broken in kind of a subtle way.

This is a protocol called Denning-Sacco. And when it was originally proposed, the original proposers of it actually got it wrong, too.

And they presumably thought about it for a while before they got it wrong.

This is meant to be an illustration that designing cryptographic protocols is hard and it needs to be something that you need to think about very carefully.

Here is the protocol.

Suppose that Alice wants to send a message between Alice and Bob and we have our certificate authority.

Alice says to the certificate authority, I would like to do some communication with Bob.

And my name is Alice.

What Bob sends back is one of these certificates, a signed message that has Alice's public key in it and Bob's public key.

We call these signed messages certificates.

These come from the certificate authority.

And they basically are a way that if Alice trusts the certificate authority she can decode Bob's public key and be reasonably assured that this is, in fact, Bob's public key.

She is also going to get a certificate for herself that she can send to Bob so that Bob doesn't have to go contact the certificate authority again.

She is going to send this thing that has been signed the certificate authority which is going to allow Bob to authenticate her.

So she does that.

What she sends to Bob is her own certificate which is this little thing shown here on the left.

And then she also sends to Bob a proposed shared key.

This is kAB.

And she signs that proposed key with her own private key.

When she signs this key with her private key that means that Bob can be assured that this is

an authentic message, in fact, from Alice.

And then she encrypts it with Bob's public key so that only Bob can be the one who can decode it.

So only Bob will actually see the contents, see what this key is.

And these Ts that I have shown here are simply timestamps, and I will explain to you why those are important in a minute.

We need timestamps that go along with these things.

So this proposed key has been both signed with her private key and encrypted with Bob's public key.

And now Bob can go ahead and send messages back to Alice.

And he can encrypt those messages using kAB, using a shared key encryption mechanism which, as we said, is more efficient.

So these guys can exchange a bunch of information with each other in this way.

This is an example of a cryptographic protocol.

I said there was a bug.

Let's try and debug it a little bit.

And, to get at that, let's talk about what some of the properties of cryptographic protocols that we would like.

One property we would like is what is called freshness.

What freshness means is that I am reasonably assured that this message is recent.

It came from recent history.

And this is what we need the timestamps for.

We need to make sure that this message is, in fact, a message that is relatively new.

It is not an old message that I generated before that is now being sent back to me again.

That is what we are going to use the timestamps here for.

We also want to make sure that the message is appropriate.

And, by appropriate, I am actually the intended recipient of this message and the sender of this message is actually who the sender of this message claimed to be.

This message actually should be applied right now.

This message makes sense.

And then, finally, the third property we want is something called forward secrecy.

What this says is essentially we should be able to change the cryptographic keys that we are using at some point in the future.

Say, for example, if our keys become compromised, we should be able to change the keys and our method should continue to work.

This approach is pretty clearly true.

Alice could request a new key.

And then you could imagine some way in which Alice could create a new key and send that new key to Bob and we can sort of start all over again.

The two properties that are really interesting are this freshness property and this appropriateness property, and they are the two properties that I want to talk about a little bit more.

And if you think about what the kinds of attacks are that somebody could apply on this thing, well, one attack they could apply would be an attack on the sort of cryptographic transform itself.

A brute force attack where you try and factor the keys, we said that is hard.

Another attack that they could apply would be a so-called replay attack.

There is a crypto attack and so-called replay attack.

This is that somebody might sort of reuse a message that they saw transmitted over the network before.

Some Eve or Lucifer might overhear a message and then might try and resend that message in order to get the server to do something.

Suppose there is a message that tells the server to take some action in the outside world, open a door, so you might save up a message that somebody else had sent before that said open the door.

And then you might resend it in order to get the door open when you wanted it to open.

We want to prevent that.

And we are going to use timestamps to do this.

And getting the timestamps to work out is a little bit tricky.

If the sender and the receiver have perfectly synchronized clocks, you might just say as long as the timestamp is within the first few milliseconds that is fine.

But if that is not true then you need to do something a little bit more sophisticated.

There is a protocol called Kerberos that works this out in detail.

And that, again, is described in the notes.

But the idea is just you put the timestamp in every message that you send.

And then when you sign the message, when the person on the other end decodes that message, they know that this message was generated as of a particular time.

But the third kind of attack we might be worried about is a so-called impersonation attack.

And this is the attack that this protocol is susceptible to.

Does anybody see what the bug in this protocol is?

STUDENT: Charles does not authenticate that Alice is Alice.

Charles does not authenticate?

Well, Alice does send this key that says -- Charles doesn't actually care because all Charles saw was a message that said I want Alice's public key and B's public key.

There is nothing wrong with that message getting corrupted.

Alice might get something back, but if it comes back it has been signed with Charles' private key and so she will know exactly what it was that was in there.

STUDENT: Only Alice knows Charles private key?

I mean can't someone else impersonate Alice and do that whole process and then be seen as Alice by Bob?

But notice that the key has been signed with Alice's private key.

This proposed key has been signed with Alice's private key.

STUDENT: Alice is sending something encoded in Charles' private key, is that right?

No.

Well, all this means is that the thing that is encoded with Charles' private key, Bob is going to be able to decode that thing using Charles' public key.

And he is going to have some assurance, in fact, about what Alice's public key is.

I think that is OK.

Anybody else want to take a guess?

OK.

Well, since we are out of time, I will talk about this the first thing next time and will show you what the problem with this protocol is.

I will see you on Monday.

One announcement before you guys go.

I have a little gift for you guys which is that we are going to cancel class next Wednesday before the design project is due.

So there is no class next Wednesday, but there is class on Monday so make sure you come here.