MITOCW | ocw-6.033-lec22

0:00:00 0:00:20 We started last time on protection. 0:00:23.852 You remember the model that we're considering? 0:00:28.951 We've got some client communicating over the Internet 0:00:34.844 with some server. And we said that there are 0:00:39.716 three main technical problems that we're going to look to 0:00:46.062 address. There is the problem of 0:00:49.575 authentication, the problem of authorization 0:00:54.447 and the problem of confidentiality. 0:01:00 0:01:05 Today what we're going to do is we're going to talk mostly about 0:01:09.221 this issue of authentication, but I want to just return to 0:01:13.041 some of the details that we were talking about at the end of 0:01:16.994 class last time so that you guys all get the important bits out 0:01:21.149 of our discussion of the RSA protocol. 0:01:23.628 Remember, last time we started talking about these are the main 0:01:27.783 requirements of a protection system. 0:01:31 And we drew a diagram that said that at the top level we want to 0:01:35.295 have an application that is protected. 0:01:37.818 Underneath that we're going to have a set of security 0:01:41.363 primitives. And then, underlying all of 0:01:43.954 this, we're going to have cryptography. 0:01:46.545 Cryptography is going to be sort of the essential technique 0:01:50.5 that we use to enable protection. 0:01:52.681 We talked briefly about several different types of cryptography. 0:01:58 Remember, we had our model where we have A is sending 0:02:02.5 through some box which takes the data and transforms it using 0:02:07.692 some cryptographic transform. And then that cryptographically 0:02:12.884 transformed data gets transmitted and then is reverse 0:02:17.384 transformed or untransformed at the other side and it comes out 0:02:22.75 to the receiver. What cryptography does is we 0:02:27.27 said that we want this open model where there is some key k1 0:02:32.054 here and some key k2 here that are used to decide how this 0:02:36.675 transform operator works and this untransform operator works 0:02:41.459 on the other side. We talked about a shared key 0:02:45.189 case where k1 is equal to k2. And the appendix A of the 0:02:49.567 chapter in the notes goes over a shared key protocol called DES 0:02:54.594 that is fairly commonly used. The basic idea behind shared 0:02:59.904 key cryptography is typically that both sides have this key 0:03:03.848 and we're going to transform the data by combining the message 0:03:07.996 stream that is coming from A together with the key in some 0:03:11.872 way, say, for example, by xoring the data in many 0:03:15.135 different ways or permuting the data around and then xoring it 0:03:19.283 with the key in some combination of ways. 0:03:22.003 That's sort of what the DES protocol does. 0:03:24.792 It's just multiple rounds, basically, of permuting and 0:03:28.395 xoring the data over and over again. 0:03:32 The idea is that if an attacker doesn't know the key, 0:03:36.197 it's very hard for them to reverse this process without 0:03:40.557 having the key, but if you know the key it's 0:03:44.028 relatively easy to reverse the process. 0:03:47.096 We also talked a little bit about public key cryptography 0:03:51.617 where k1 is not equal to k2. In this case, 0:03:54.927 we started talking about this protocol RSA which is an example 0:03:59.851 of a public key protocol. If you remember, 0:04:04.046 RSA has some set of rules for actually doing the 0:04:07.121 transformation of data. The basic idea is that we're 0:04:10.457 going to get a private key and a public key here, 0:04:13.598 which is going to consist of this prime number plus the 0:04:17.13 product of these two prime numbers and this private key k 0:04:20.794 which is going to consist of the other prime number and the 0:04:24.588 product of these two things. Now we have this way in which 0:04:28.317 we can transform the data. What this transform box does is 0:04:32.833 simply raise the message to the power of e modulo n and we have 0:04:36.5 the reverse transform. This is just a specific 0:04:39.161 mathematical operation. The reason that this 0:04:41.704 mathematical operation works is that the intuition is that it is 0:04:45.43 relatively easy to take something to an exponent. 0:04:48.268 I can take a number and exponentiate it without doing 0:04:51.344 very much work. But in order for an attacker to 0:04:55.174 compromise this system, that is in order to discover, 0:04:58.821 for example, what the private key is, 0:05:01.347 in this case, what this value d is, 0:05:03.732 given just the public key which consists of e and n, 0:05:07.309 the attacker is going to be able to factor this number n, 0:05:11.237 which turns out to be very difficult to do. 0:05:14.183 Essentially, breaking this cryptographic 0:05:16.919 system comes down to be able to factor a very large number. 0:05:20.987 And that turns out to be, computationally, 0:05:23.863 very, very difficult to do. So that's sort of the intuition 0:05:27.931 behind the protection of this scheme. 0:05:32 The nice thing about RSA is it has, in most public key 0:05:35.54 cryptographic protocols is that they have this property that the 0:05:39.75 reverse is also true. So I can transform using 0:05:42.756 exponentiation with the public key or the private key and I can 0:05:46.898 reverse transform using the opposite. 0:05:49.303 If I transform with the public key, I can reverse transform 0:05:53.178 with the private key and vise versa. 0:05:55.517 So we are going to exploit this property a lot when we're 0:05:59.258 talking about this kind of cryptography. 0:06:03 Just to sort of point out that this is hard, 0:06:05.734 this is an estimate from the RSA Security Company about how 0:06:09.423 many compute years on modern computers it would take to 0:06:12.857 factor an RSA key of a given link. 0:06:14.956 So the bottom is the key link in bytes on the x-axis and then 0:06:18.771 the y-axis is the machine years to factor. 0:06:21.379 If the machine years to factor was 1,000, that means it would 0:06:25.195 take 1,000 machines working together one year to do this or 0:06:28.883 one machine 1,000 years to do it. 0:06:32 So you see that the y-axis here is going up exponentially. 0:06:36.035 So to factor a 500 byte key is not that hard. 0:06:39.15 It takes maybe a few months of compute time to do it. 0:06:42.831 To factor a 2,000 byte key takes this a ridiculously long 0:06:46.796 time, ten to the 16th years it is going to take to factor this 0:06:51.115 thing. So there is just no way that, 0:06:53.592 at least modern computers, are ever going to be able to 0:06:57.415 actually perform this factoring using currently known algorithms 0:07:01.876 for factoring. It is, of course, 0:07:04.959 possible that somebody will develop a very fast factoring 0:07:08.118 algorithm or that there will be some fundamental breakthrough 0:07:11.503 like quantum computing that will make it possible to factor very 0:07:15.058 large numbers. But, at this point, 0:07:16.92 this is sort of the estimate of how long it would take to do 0:07:20.249 this. So, by using long keys, 0:07:21.828 you can be sort of guaranteed that somebody won't be able to 0:07:25.157 actually factor this number. So by using a factoring-based 0:07:28.373 attack on RSA that sort of won't be able to break it in this way. 0:07:33 The largest

factoring based on Fermat's attack on RSA that sort of lets be able to break it in this way. The largest RSA number that has been factored to date is some 0:07:36.866 500 odd byes, something like 520 bytes, 0:07:39.315 so RSA has challenges for the largest numbers that can be 0:07:42.923 factored. So we're sort of just barely on 0:07:45.501 this curve is where the largest numbers that have been factored 0:07:49.496 to date. And these are typically some 0:07:51.816 collection of machines on the Internet. 0:07:54.264 Somebody gets 1,000 machines on the Internet and they chew on 0:07:58.131 this problem. I think, in this case, 0:08:01.896 it took 1,000 machines three months to find a factor of this 0:08:06.982 500 byte key. This is the basic idea now. 0:08:10.431 What we're going to do is use these cryptography algorithms, 0:08:15.517 this math to build up a set of primitives that we can use to 0:08:20.603 actually accomplish our security goals that we want. 0:08:25 And I put these on the board briefly last time. 0:08:30 But I didn't talk at all about what the actual functionality 0:08:32.892 is. And we're going to spend a lot 0:08:34.509 of time talking about two of these primitives today, 0:08:37.009 and then we'll talk about a few more of them next time. 0:08:40 0:08:48 We have two sets of primitives in particular that we are 0:08:54.226 interested in. One set is called sign and 0:08:58.754 verify. And the other set is called 0:09:01.643 encrypt or decrypt. And we're going to use sign and 0:09:04.685 verify basically as a way to achieve authentications. 0:09:07.849 Remember, authentication is the process of verifying that a 0:09:11.379 message actually came from who it was supposed to come from. 0:09:14.969 And we're going to use encrypt and decrypt in order to provide 0:09:18.681 confidentiality to guaranty that somebody who we don't want to be 0:09:22.576 able to read a message, somebody who shouldn't be able 0:09:25.801 to read a message can, in fact, not read that message. 0:09:30 Let me just illustrate a simple example. 0:09:35.282 Suppose we're using a public key system, we can say suppose 0:09:43.139 we use something like RSA to encode a message m with some 0:09:50.725 key, say we're sending a message from A to B, we use it to encode 0:09:59.394 some key, say, kA private. 0:10:04 What this notation means is that whoever has A's private key 0:10:09.344 encodes that message or transforms that message using a 0:10:14.236 cryptographic protocol using, say, for example, 0:10:18.404 kA private. What this says is Alice, 0:10:21.574 perhaps as A, runs her RSA algorithm on the 0:10:25.379 message and generates some encoded thing. 0:10:30 This is an encryption. We're going to use this 0:10:34.368 encoding with the private key to sign a message. 0:10:38.932 When Alice encodes this with her private key what that means 0:10:44.66 is that somebody who has Alice's public key can decode this, 0:10:50.388 because we said if you encode with the private key somebody 0:10:56.019 with the public key can decode. For example, 0:11:00.675 if she sends this to Bob, Bob is going to be able to 0:11:03.545 decode this. At first that seems like, 0:11:05.628 well, what good does that do us? 0:11:07.373 Bob can read this message. But the thing is if Bob can 0:11:10.356 decode this, what he knows, with very high certainty is 0:11:13.395 that the person who encoded this message had access to A's 0:11:16.604 private key. If he is willing to trust that, 0:11:19.024 in fact, the only person who has access to A's private key is 0:11:22.401 A herself, then he can be reasonably confident that this 0:11:25.497 message came from A. You guys all sort of see how 0:11:28.198 that works. What we say is that Alice signs 0:11:33.069 this message. And, by signing this message, 0:11:37.227 she is able to sort of give Bob some confidence that this 0:11:42.772 message, in fact, came from her. 0:11:45.841 We can also do the opposite thing where, say, 0:11:50.198 for example, Alice is sending a message to 0:11:54.257 Bob. She can take m and sign it with 0:11:57.722 kB public. Sorry, she can encrypt it with 0:12:01.99 kB public. She is going to use the RSA 0:12:04.444 algorithm. She is going to encrypt this 0:12:06.965 message. And, when she encrypts this 0:12:09.287 message, she is going to basically use the RSA algorithm 0:12:12.936 using kB public. And what is going to happen is 0:12:15.987 now she is going to be reasonably assured that this 0:12:19.304 message is going to be able to only be read by Bob because only 0:12:23.417 Bob presumably has his own private key. 0:12:27 What I've shown here makes this look very simple. 0:12:29.884 I sort of said it as though we're simply taking this message 0:12:33.43 and running the basic RSA algorithm, as I showed it, 0:12:36.495 on the message in order to do sign and in order to encrypt. 0:12:39.981 It turns out that actually getting sign and encrypt to work 0:12:43.467 property requires some fairly sophisticated sort of reasoning 0:12:47.073 about the RSA algorithm. For example, 0:12:49.237 when you're encrypting something, if the message is 0:12:52.242 very small, it turns out that if you encrypt a small message 0:12:55.787 simply using the RSA algorithm as I presented it that it's 0:12:59.213 relatively easy to break the RSA algorithm. 0:13:03 The RSA algorithm becomes vulnerable with very small 0:13:05.684 messages. That means the encryption 0:13:07.473 process actually needs to pad out the message to some longer 0:13:10.578 link. It needs to put unused bytes at 0:13:12.473 the end of every message in order for it to be secure. 0:13:15.263 These sign and encrypt algorithms themselves sort of 0:13:17.947 have to do some additional work on top of the basic RSA 0:13:20.789 algorithm in order to be secure. And that additional work is 0:13:23.894 sort of beyond the scope of this class. 0:13:25.894 It requires a certain degree of mathematical sophistication that 0:13:29.21 we are not going to rely on here. 0:13:32 You can go take an extra class about cryptography and learn 0:13:36.441 about how these sign and encrypt algorithms actually work, 0:13:40.806 but the basic idea is that you can sort of grasp it by just 0:13:45.247 seeing RSA. These things are hard to build. 0:13:49 0:13:54 Now that we have sort of quickly reviewed the basics of 0:13:58.712 protection and we've seen the sign and verify and encrypt and 0:14:03.949 decrypt primitives, now what I want to do is to 0:14:07.963 talk a little bit about how we can actually start building up, 0:14:13.287 solving this problem of authenticating a user. 0:14:18 When we authenticate a user we have two objectives. 0:14:24.746 One thing we're trying to achieve is to determine who is 0:14:32.166 making a request. 0:14:35 0:14:40 In practice, it's going to turn out, 0:14:43.173 in fact, that we may not officially be able to say 0:14:47.616 exactly the specific person who is being able to make the 0:14:52.694 request. We may only be able to say that 0:14:56.23 the same principle or person or computer as before is making the 0:15:01.943 request, or the same principle who had some special privileged 0:15:07.474 piece of information that the person before had is making this 0:15:13.005 request. We may not be able to actually 0:15:16.453 associate this with a physical person, but at least we're going 0:15:19.456 to be able to know that we've seen somebody who had this same 0:15:22.363 information before. We're trying to make sure that 0:15:24.737 this is sort of consistent with a series of requests that we've 0:15:27.74 seen in the past. A simple

example is I lot onto 0:15:31.399 Amazon, I create a password. Now, anybody who I give that 0:15:35.879 password to can log onto Amazon. And all the system can say is 0:15:40.759 that it knows that whoever it is who is accessing definitely has 0:15:45.799 my password information and sort of it doesn't know officially 0:15:50.679 that it is actually the same physical person. 0:15:55 The other thing, though, that we want 0:15:57.58 authentication to do is to tell us basically that the message 0:16:01.881 that was sent is equal to the message that was received. 0:16:05.824 We want it to be the case that if Alice sends a message to Bob, 0:16:10.268 that when Bob receives that message, we want him to have 0:16:14.211 some assurance that, in fact, that message was 0:16:17.437 actually the message that Alice tried to send. 0:16:20.663 And we're going to talk about how this is sort of going to 0:16:24.749 relate to the problem of integrity that we talked about 0:16:28.62 before, verifying that a message hasn't been corrupted in 0:16:32.634 transmission. But this is actually a harder 0:16:36.625 requirement because we might need to not only worry about 0:16:39.762 sort of random bits being flipped by the communication 0:16:42.732 channel but we need to worry about some malicious person 0:16:45.814 coming in and physically actively modifying the bytes of 0:16:48.896 the message. Authentication is sort of the 0:16:51.193 first step that we need to have in any secure system because 0:16:54.5 authentication is going to be the key. 0:16:56.573 We have to authenticate a user before we can authorize the 0:16:59.767 user. Before we can say, 0:17:02.211 yes, this user is allowed, we have to figure out who the 0:17:05.911 user is. And, similarly, 0:17:07.459 we can use authentication, this process of figuring out 0:17:11.092 who a user is to help us to build up audit trails. 0:17:14.389 Remember we said audit trails are something that is important. 0:17:18.494 Once we figure out who a user is, we can keep logs of which 0:17:22.396 user it was that connected to our machine. 0:17:25.155 Let's look at a simple model of how authentication works. 0:17:30 0:17:35 The idea is as follows. We have our principle, 0:17:41.83 say, for example, a user who is sending a request 0:17:49.116 to some server. 0:17:52 0:17:57 And this request might be something, for example, 0:18:00.263 like buy Apple stock. And the server's job is to 0:18:03.46 determine whether this request is actually authentic, 0:18:06.996 whether it actually came from the user it claims to come from 0:18:11.076 and whether the message that was received is actually the message 0:18:15.428 that was sent. The way it's going to do this 0:18:18.352 is we're going to introduce something we call a guard that 0:18:22.228 is at the front end of this server that takes in all the 0:18:25.968 incoming requests and is in charge of authenticating them. 0:18:31 And then this guard is going to dispatch results off to any of 0:18:34.714 the services that the server provides. 0:18:36.967 It is going to say if the request is authentic it is going 0:18:40.437 to invoke the service that we need. 0:18:42.508 In other words, this guard is the type of 0:18:44.943 mediator that sits in between all the requests between the 0:18:48.414 principle and the service that it is trying to access and the 0:18:52.067 guard is going to be the thing that does all of our 0:18:55.112 authentication. 0:18:57 0:19:07 Now the question is what is it exactly that the guard does? 0:19:10.914 What are the processes that the guard has to go through in order 0:19:15.167 to authenticate a user? And, before we talk about the 0:19:18.677 specific technical solution, I want to sort about two things 0:19:22.66 that are really going on when you're trying to figure out who 0:19:26.71 it is that is requesting and whether you're going to allow 0:19:30.557 that person to do what they want to do. 0:19:34 So the issue is, essentially, 0:19:36.044 as follows. There is this question about is 0:19:39.112 the computer that is making this request actually have the 0:19:43.275 appropriate keys to make this request? 0:19:45.977 Does it have the appropriate password to be allowed to make 0:19:50.213 this request? Then there is this other 0:19:52.915 question which is do I trust this user? 0:19:55.691 Is this somebody who I want using my service? 0:20:00 Suppose that I go to E*Trade and say I want to buy Apple. 0:20:04.137 How does E*Trade decide whether or not it trusts me, 0:20:07.905 whether I'm somebody who is allowed to buy Apple stock from 0:20:12.189 them or not? So they have some process for 0:20:15.218 doing that. You have this mechanical 0:20:17.804 process of authentication. And then we have this sort of 0:20:21.868 psychological process of determining whether or not you 0:20:25.857 trust somebody. So this authentication is going 0:20:30.896 to be a technical thing and we are going to describe this as 0:20:36.409 being psychological. What these things are connected 0:20:41.174 by is some notion of a name. What the authentication says 0:20:46.406 is, for example, Sam wants to buy Apple. 0:20:50.05 It determines that I am, in fact, the person who is 0:20:54.722 making the request, that it is able to verify that 0:20:59.3 I am the one who is actually making the request. 0:21:05 What trust is responsible of doing is determining whether or 0:21:08.485 not Sam should actually be allowed to make this request. 0:21:11.734 And this is not something that we necessarily have a technical 0:21:15.337 solution to. Different computer systems have 0:21:17.877 different ways of determining whether or not they should trust 0:21:21.481 each other. For example, 0:21:22.839 when you go to a service on the Web, you make a decision about 0:21:26.443 whether you trust that service based on some sort of way that 0:21:29.987 you have of evaluating the service. 0:21:33 Maybe your Mom told you that E*Trade is the best place to buy 0:21:36.157 stocks on the Internet with. You trust your Mom and so, 0:21:39 therefore, you trust E*Trade. Or maybe you saw an 0:21:41.526 advertisement for E*Trade and you think any company that has 0:21:44.631 enough money to advertise on the side of the freeway must be a 0:21:47.842 trustworthy company and, therefore, I trust you. 0:21:50.315 Similarly, E*Trade, presumably determines that they 0:21:52.947 trust you based on, well, one thing they might do 0:21:55.473 is, you might give them a bank account number, 0:21:57.842 and your bank might say, yes, this person has enough 0:22:00.526 money to make this request. That is a way that they might 0:22:04.788 decide that they trust you. Alternatively, 0:22:07.315 E*Trade gives out loans, so maybe they go talk to your 0:22:10.583 credit reporting agency, and the credit reporting agency 0:22:13.974 says, yeah, this guy is trustworthy. 0:22:16.132 He has a high credit score so, therefore, E*Trade will let you 0:22:19.894 borrow $1,000 to invest in the stock market or something. 0:22:23.346 This issue of trust is one that we don't have a technical 0:22:26.799 solution to, but it is absolutely an essential part of 0:22:30.067 authenticating and authorizing a user to use a computer system. 0:22:35 So it's worth just bearing this in mind that any time you're 0:22:38.859 building a computer system you need to think about how it is 0:22:42.719 that you determine whether a user is trustworthy or not. 0:22:46.317 And this goes both ways, both

for the user of the 0:22:49.457 system, how do they determine that you're trustworthy, 0:22:52.925 and for the system, how do you determine that the 0:22:56.065 users are trustworthy? Now the key problem is focusing 0:22:59.532 on this sort of authentication problem. 0:23:03 What is the technical meat of how we're actually going to 0:23:05.871 authenticate a user? 0:23:07 0:23:17 One of these problems is determining that this message 0:23:21.373 that was sent was actually the message that was received. 0:23:25.995 Here is a simply proposal that I might have which is I might 0:23:30.864 compute one of these CRC checks. These checksums that we talked 0:23:35.693 about when we were talking about networking that we used to 0:23:38.967 determine whether or not the message has been corrupted in 0:23:42.185 transmission. It turns out that is not going 0:23:44.612 to work. You might also think the way 0:23:46.645 that you are going to insure that the person who is 0:23:49.467 requesting the message is actually who they claim to be 0:23:52.516 is, for example, using our encryption primitive. 0:23:55.169 If we had a one time pad available to us, 0:23:57.427 we might encrypt the entire stream of data that is going 0:24:00.532 across. And then we would think well, 0:24:04.1 there is no way that somebody on the other end, 0:24:07.432 unless they are authorized to read this, will actually be able 0:24:11.85 to read it. And, therefore, 0:24:13.733 this is going to give us some guaranty about who is making the 0:24:18.151 request. Neither of these things quite 0:24:20.831 works, and let me explain why. Integrity, in the sense that we 0:24:25.25 studied it previously in 6.033, is not equal to authenticity, 0:24:29.595 is not equal to confidentiality. 0:24:33 0:24:38 And the reason is as follows. Let's just look at a simple 0:24:42.2 example. Suppose there is Alice who is 0:24:44.974 sending a message to some server, and suppose this server 0:24:49.174 is something where Alice doesn't really mind if her request is 0:24:53.75 seen in public but she wants to make sure that the server 0:24:57.95 actually receives the request that she made. 0:25:02 So she cares about it being the right request, 0:25:05.425 but she doesn't care about it being a secret request. 0:25:09.383 Maybe Alice says I want to donate $100 to Save the Whales. 0:25:13.721 She says I don't care if anybody knows that I want to 0:25:17.679 save the whales. That's a noble and respectful 0:25:21.104 cause. In fact, I am proud of the fact 0:25:23.921 that I want to save the whales. So anybody who wants to can 0:25:28.335 look at this message. But I really only want to give 0:25:33.059 Save the Whales $100. I want to make sure that 0:25:36.148 somebody doesn't mess with this message and some Save the Whale 0:25:40.405 activist doesn't change this message to be $10,000 so that I 0:25:44.455 don't lose a bunch of money. Let's look at some techniques 0:25:48.368 that we might use to sort of guaranty that this is the case. 0:25:52.418 One thing we might do is, for example, 0:25:54.958 use a one time pad. Suppose Alice and the server 0:25:58.185 share a random bit string. Alice might encode, 0:26:02.283 might xor the message that she transmits, and then the server 0:26:06.337 might apply the xor again with the same bit string and get the 0:26:10.459 message back. The problem with this is that 0:26:13.297 this doesn't prevent some malicious user or Lucifer from 0:26:17.013 coming into the middle of this and interfering with this byte 0:26:21.067 string as it is transmitted. Even though Lucifer cannot 0:26:24.716 actually look at the message, he can manipulate the byes to 0:26:28.635 his heart's content. And he may get lucky and change 0:26:33.296 the bytes so that it says something like donate some 0:26:37.074 different amount of money to save the whales or he may just 0:26:41.37 garble Alice's requests so it cannot be processed. 0:26:45 So one time pad is not going to work. 0:26:47.666 Now, the other alternative I said we can talk about would be 0:26:52.037 something like using a CRC. What Alice would do is send m 0:26:56.185 followed by the CRC of m. The problem with this is that 0:27:00.644 now Lucifer can look at this message as it comes across the 0:27:03.932 wire and the checksum, and he can simply change the 0:27:06.768 message to say whatever he wants and then recomputed the 0:27:09.886 checksum. So the checksums, 0:27:11.36 as we have talked about them so far, are good for handling the 0:27:14.819 case of non-malicious attacks or non-malicious problems like a 0:27:18.278 byte gets changed in transmission or there is some 0:27:21.056 error in transmission, but they don't solve this 0:27:23.721 problem of preventing somebody from tampering with the message 0:27:27.18 and being able to detect when that tampering also effects the 0:27:30.582 CRC. If you think about this for a 0:27:34.402 minute, the property that we want sort of feels like some 0:27:39.041 combination of these two things. Basically, what we want is some 0:27:44.26 checksum-like operation such that we know that the only 0:27:48.733 person who could have possibly written this checksum was 0:27:53.289 somebody who access to some key that Alice has like Alice's 0:27:58.094 private key. We want a checksum that is 0:28:02.314 dependent on a key, and we want this checksum to 0:28:06.497 have some fairly strong properties. 0:28:09.523 We want this checksum to be resistant to these kinds of 0:28:14.329 attacks that we talked about Lucifer applying. 0:28:18.335 In particular, we want the checksum to be 0:28:21.895 resistant to Lucifer modifying the message. 0:28:25.633 So the checksum should detect that this message was modified, 0:28:30.973 if it was modified. We also want it to be able to 0:28:36 detect other transformations of the message. 0:28:38.965 For example, suppose that rather than 0:28:41.448 modifying m, changing something, Lucifer just reorders some of 0:28:45.655 the words in it, keeps exactly the same 0:28:48.275 characters and exactly the same length but just switches some 0:28:52.413 things around. And there are other kinds of 0:28:55.31 things that we might want to be able to resist as well. 0:29:00 For example, taking exactly the same message 0:29:02.648 and sticking a little bit of extra data onto the end of it, 0:29:06.221 appending something to the message. 0:29:08.316 There is a bunch of different sort of properties that we want 0:29:12.012 whatever this thing is that is going to allow us to do 0:29:15.277 authentication to have. Let's look at the sort of model 0:29:18.603 for how our basic authentication system is going to work. 0:29:22.053 It is really pretty straightforward and is going to 0:29:25.133 rely on using our sign and verify primitives that I am 0:29:28.398 erasing. 0:29:30 0:29:35 The idea is as follows. We have Alice over here that is 0:29:39.72 sending a message. That message is going to go 0:29:43.653 into our sign cryptographic primitive, it's going to be 0:29:48.374 transformed and is going to come over here to our verify box, 0:29:53.618 and then it is going to come out on the other side at the 0:29:58.513 server. And the sign is going to have 0:30:01.626 some k1 that gets fed into it and verify is going to have some 0:30:05.046 k2 that gets fed into it. And the idea is that in 0:30:07.738 addition to signing the message we are also going to transmit 0:30:11.102 the message itself. So this is going to be sort of 0:30:13.85 like this idea

of putting a CRC on a message, 0:30:16.317 but this thing that we are going to do when we sign is 0:30:19.289 going to provide this sort of better properties than the basic 0:30:22.71 CRC did. The idea is this is a small bit 0:30:25.493 of additional information that gets transmitted with the 0:30:28.144 message that allows the server to be confident that, 0:30:30.602 in fact, the message that was transmitted came from Alice and 0:30:33.493 that it has not been corrupted along the way. 0:30:36 0:30:46 This is the basic idea now k1 comes in, it gets transmitted, 0:30:49.771 goes through the verify box, and the verify box reports yes 0:30:53.479 or no, this message was correct or was not correct. 0:30:56.675 And the server also has access to the message. 0:31:00 If the server gets yes in the message then it continues to 0:31:04.51 process it. And we have two kinds of ways 0:31:07.676 of doing authentification that correspond to shared key and 0:31:12.266 public key cryptography. If k1 is equal to k2 we say 0:31:16.302 that this little bit of information that we have 0:31:20.021 transmitted is a MAC, a message authenticator. 0:31:23.582 When you are using shared key cryptography that is the bit of 0:31:28.33 additional information that gets transmitted. 0:31:33 And when k1 is not equal to k2, that is if we're using public 0:31:37.792 key cryptography, we call this bit of additional 0:31:41.547 information a signature. To give you a very simple 0:31:45.461 example of what a signature might consist of, 0:31:48.976 suppose we have a message m. In public key cryptography, 0:31:53.369 a simple kind of signature that you can compute is to take the 0:31:58.242 hash of m. And then take that thing and 0:32:02.462 encode it with Alice's private key. 0:32:05.453 This is a simple kind of a signature that we might attach 0:32:10.379 to a message. And this has the properties we 0:32:14.162 want, which is that we believe that only Alice could have 0:32:19.087 actually encoded this message because we belief that only 0:32:24.013 Alice has access to her private key. 0:32:27.092 And so when somebody uses the public key to decode this they 0:32:32.282 get something back which is a hash of the message. 0:32:38 And we call this a cryptographically secure hash. 0:32:43.608 It is some way of combining the bits of the message together 0:32:50.501 such that when the server computes the hash of m, 0:32:56.109 the probability that it matches this hash that was in the 0:33:02.652 signature that Alice sent, they will match if the messages 0:33:09.312 are the same. And, if the hash was computed 0:33:13.689 on a different message than the message that the server 0:33:16.834 receives, the probability is very, very low that these two 0:33:20.155 messages are, in fact, the same. 0:33:21.961 So they say it has a low collision probability. 0:33:24.64 It's very unlikely that some m prime that is not equal to m has 0:33:28.252 the same hash as m. That's what a cryptographically 0:33:31.815 secure hash provides us. And, again, cryptographically 0:33:34.92 secure hashes are another kind of sort of mathematical 0:33:38.025 technique. There is a set of algorithms 0:33:40.251 for driving these things. The appendix talks about a 0:33:43.238 common one which is used called sha-1. 0:33:45.405 And sha-1, it turns out, has recently been shown to have 0:33:48.627 some higher probability of collisions than was previously 0:33:51.907 thought. And there is a new protocol 0:33:53.958 called sha-256 which is still believed to be secure. 0:33:58 But sha-1, the S-H-A, which stands for secure hash, 0:34:01.161 is one that is very commonly used in the world today. 0:34:04.449 So this is a simple example of a signature and it allows 0:34:07.926 basically for the server to verify one by decoding the 0:34:11.277 message with Alice's public key that the message actually came 0:34:15.134 from Alice, and this hash allows it to be reasonably assured that 0:34:19.181 the message has not been tampered with. 0:34:21.584 If the message had been tampered with the hash wouldn't 0:34:24.998 match, and if this signature had been tampered with then the 0:34:28.728 receiver wouldn't have been able to decode this thing and get 0:34:32.522 something back that made any sense. 0:34:36 Or, again, the hash probably wouldn't match. 0:34:40.909 This is the basic solution for doing authentication. 0:34:46.732 This screen is too low. Now that we've sort of seen a 0:34:52.669 method for authenticating a message, there are sort of a 0:34:58.948 bunch of little details that we left hanging. 0:35:05 In particular, one of the details that we 0:35:07.636 haven't yet addressed is, suppose we are using some 0:35:10.931 public key based mechanism like this, I haven't yet told you or 0:35:15.017 we haven't talked about how it is that somebody would learn 0:35:18.839 Alice's public key? We sort of just assumed that 0:35:21.936 whoever is receiving the message is able to figure out what 0:35:25.759 Alice's public key is. And one thing you could imagine 0:35:29.637 is that Alice goes around and physically meets everybody who 0:35:32.755 she would ever want to give a key to and she hands them a slip 0:35:35.978 of paper that has her public key written on it. 0:35:38.409 But, obviously, one, that's not how the 0:35:40.416 Internet works clearly because we have some cryptographic 0:35:43.375 protocols that allow us, we don't have to meet everybody 0:35:46.281 and we can still get established secure communication channels. 0:35:49.557 And, two, that just doesn't sound very scalable. 0:35:52.041 It doesn't sound like a great solution. 0:35:55 This is the key distribution problem. 0:35:57.495 This is going to apply mostly for the case of public key 0:36:01.306 cryptography. In the case of shared key 0:36:03.94 cryptography, this method I am going to show 0:36:06.92 you in a minute is not going to apply necessarily to shared key 0:36:11.217 cryptography, but I will show you a way in 0:36:14.059 which we can use public key cryptography to sort of 0:36:17.524 bootstrap or to exchange a shared key which we can then use 0:36:21.544 over our communication channel. This is the key distribution 0:36:25.633 problem for public key cryptographic systems. 0:36:30 0:36:35 One solution, if Alice cannot physically meet 0:36:40.617 with Bob, one thing that you might imagine doing is you could 0:36:48.276 just have Alice send a message to Bob saying here is my public 0:36:56.063 key. Or, better yet, 0:36:58.489 Alice's public key is, A's public key is X. 0:37:05 But this isn't a very good approach because how does B 0:37:08.256 actually know that this was Alice that sent this message? 0:37:11.698 This could have been some Lucifer who said, 0:37:14.279 oh, by the way, Alice's key is this. 0:37:16.43 And now B thinks that Alice's key is, in fact, 0:37:19.195 Lucifer's key. And now Lucifer can decode any 0:37:21.899 message that B wants to send to A. 0:37:23.927 So this isn't really a very good approach because if A just, 0:37:27.553 sort of out of the blue, blurts out to be my key is this 0:37:30.932 then it doesn't have any way of actually knowing that this was, 0:37:34.743 in fact, A who reported this key. 0:37:38 That doesn't solve our problem. Instead, we are going to use a 0:37:41.279 technique called certificates. 0:37:43 0:37:48 Certificates work like this. The idea with certificates is 0:37:52.576 that we are going to introduce a third-party who I have shown 0:37:57.394 here as Charles. Let's suppose that Alice and

0:38:01.508 Bob know and trust Charles. So they have already somehow 0:38:06.436 exchanged information with Charles, exchanged keys with 0:38:11.274 him. The idea is now suppose Alice 0:38:14.231 wants to send a message to Bob. What she does is she sends this 0:38:19.786 message, and she signs the message using her private key. 0:38:24.803 That is what this S parentheses m kApriv means. 0:38:30 Now what Bob is going to do is say, well, I don't know who this 0:38:33.419 Alice is. And rather than simply asking 0:38:35.516 Alice for her public key, which we have already said is 0:38:38.494 not a very good idea, what Bob does is he asks 0:38:40.976 Charles for her public key because he trusts Charles and he 0:38:44.176 knows that Charles probably knows something about Alice. 0:38:47.209 And what Charles responds with is Alice's public key, 0:38:50.078 as well as he needs to make sure that he, 0:38:52.284 himself, signs this message with his own private key so that 0:38:55.539 somebody else cannot intercept the message as it is coming back 0:38:58.959 and overwrite it. He put his signature on this 0:39:03.036 message that he sends back with Alice's public key in it. 0:39:07.109 Now Bob has Alice's public key, and he can go ahead and decode 0:39:11.545 this message, he can go ahead and verify this 0:39:14.745 message that Alice tried to send to him. 0:39:17.581 So, in this, we call this Charles here, 0:39:20.345 sometimes it is called a CA, a certificate authority. 0:39:25 0:39:32 And the idea is that these CAs, a small number of CAs can 0:39:35.547 service a very large number of users. 0:39:37.828 So rather than having to have every user exchange keys with 0:39:41.502 every other user, each user just exchanges keys 0:39:44.416 with a few certificate authorities. 0:39:46.57 And then, those certificate authorities, sort of act like 0:39:50.117 these hubs that propagate keys out into the rest of the users. 0:39:53.981 This is a very simple way in which you might be able to 0:39:57.402 disseminate a public key. We will talk a little bit later 0:40:02.303 on about sort of a more formal way of reasoning about how 0:40:06.464 public keys are disseminated and how these kinds of webs of trust 0:40:11.22 are built up, but this is just a simple way 0:40:14.341 to sort of start thinking about how public keys might be 0:40:18.428 disseminated automatically over the Internet. 0:40:21.698 What I want to do now is just briefly turn, 0:40:24.819 with the last few minutes, to this question of how we 0:40:28.683 actually establish a secure communication channel between 0:40:32.845 two parties. This is going to get at our 0:40:37.548 property, the sort of establishing a confidential 0:40:42.08 communication channel. We are going to talk a little 0:40:46.895 bit about how we establish confidentiality. 0:40:50.86 And so the idea is that we want to find some way that we can 0:40:56.43 encrypt the communication between our two parties. 0:41:02 And what we are going to do is first authenticate the user 0:41:10.221 using a technique like this, and then what we are going to 0:41:18.442 do is use public key to authenticate. 0:41:23.634 This is establishing a secure communication channel. 0:41:32 We are going to use public key to authenticate. 0:41:34.923 But what often happens in practice in the Internet is that 0:41:38.546 you use public key to authenticate, 0:41:40.707 and then what you are going to do is use a shared key 0:41:44.012 cryptography protocol to actually encrypt the information 0:41:47.572 that is flowing back and forth between the two parties. 0:41:51.004 And the reason for that is if you sort of looked at, 0:41:54.245 even though I said that these public key methods like RSA, 0:41:57.868 it's easier to exponentiate than it is to factor. 0:42:02 It turns out that exponentiation in RSA is still 0:42:04.396 relatively expensive because exponentiation requires, 0:42:07.048 these keys are very long, they are thousands of bytes 0:42:09.7 long, so you are taking these long messages and taking them to 0:42:12.811 very large powers. Doing that is computationally 0:42:15.208 expensive. You get these very big numbers 0:42:17.248 that computers are not terribly good at handling. 0:42:19.696 And so it has a big impact on performance if you use public 0:42:22.654 key cryptography all the time. Instead, often times what is 0:42:27.184 done is we use public key cryptography to sort of, 0:42:31.008 as I said, bootstrap the process of exchanging a shared 0:42:35.221 key between two servers. That is what we are going to 0:42:39.279 see. We are going to see how we can 0:42:41.932 then exchange a shared key which we can then use to encrypt the 0:42:46.77 communication between two parties. 0:42:49.344 This is a little puzzle for you guys. 0:42:52.154 What I am going to put up now is a broken protocol and see if 0:42:56.835 you can figure out, in the next few minutes, 0:43:00.19 how it is broken. It is not very broken but it is 0:43:04.505 broken in kind of a subtle way. This is a protocol called 0:43:07.224 Denning-Sacco. And when it was originally 0:43:09.167 proposed, the original proposers of it actually got it wrong, 0:43:12.08 too. And they presumably thought 0:43:13.586 about it for a while before they got it wrong. 0:43:15.771 This is meant to be an illustration that designing 0:43:18.151 cryptographic protocols is hard and it needs to be something 0:43:21.016 that you need to think about very carefully. 0:43:23.104 Here is the protocol. Suppose that Alice wants to 0:43:25.435 send a message between Alice and Bob and we have our certificate 0:43:28.494 authority. Alice says to the certificate 0:43:32.226 authority, I would like to do some communication with Bob. 0:43:36.602 And my name is Alice. What Bob sends back is one of 0:43:40.44 these certificates, a signed message that has 0:43:43.818 Alice's public key in it and Bob's public key. 0:43:47.272 We call these signed messages certificates. 0:43:50.496 These come from the certificate authority. 0:43:53.644 And they basically are a way that if Alice trusts the 0:43:57.636 certificate authority she can decode Bob's public key and be 0:44:02.165 reasonably assured that this is, in fact, Bob's public key. 0:44:08 She is also going to get a certificate for herself that she 0:44:11.959 can send to Bob so that Bob doesn't have to go contact the 0:44:15.849 certificate authority again. She is going to send this thing 0:44:19.877 that has been signed the certificate authority which is 0:44:23.563 going to allow Bob to authenticate her. 0:44:26.156 So she does that. What she sends to Bob is her 0:44:30.359 own certificate which is this little thing shown here on the 0:44:35.331 left. And then she also sends to Bob 0:44:38.28 a proposed shared key. This is kAB. 0:44:41.146 And she signs that proposed key with her own private key. 0:44:45.865 When she signs this key with her private key that means that 0:44:50.837 Bob can be assured that this is an authentic message, 0:44:55.219 in fact, from Alice. And then she encrypts it with 0:44:59.874 Bob's public key so that only Bob can be the one who can 0:45:03.43 decode it. So only Bob will actually see 0:45:05.951 the contents, see what this key is. 0:45:08.149 And these Ts that I have shown here are simply timestamps, 0:45:11.834 and I will explain to you why those are important in a minute. 0:45:15.777 We need timestamps that go along with these things. 0:45:19.01 So this proposed key has been both signed with her private key 0:45:22.953 and encrypted with Bob's public key.

this proposed key has been both signed with her private key of course and encrypted with Bob's public key. 0:45:25.216 And now Bob can go ahead and send messages back to Alice. 0:45:30 And he can encrypt those messages using kAB, 0:45:33.049 using a shared key encryption mechanism which, 0:45:36.241 as we said, is more efficient. So these guys can exchange a 0:45:40.354 bunch of information with each other in this way. 0:45:43.758 This is an example of a cryptographic protocol. 0:45:47.021 I said there was a bug. Let's try and debug it a little 0:45:50.851 bit. And, to get at that, 0:45:52.553 let's talk about what some of the properties of cryptographic 0:45:56.808 protocols that we would like. 0:46:00 0:46:05 One property we would like is what is called freshness. 0:46:09.545 What freshness means is that I am reasonably assured that this 0:46:14.68 message is recent. It came from recent history. 0:46:18.552 And this is what we need the timestamps for. 0:46:22.171 We need to make sure that this message is, in fact, 0:46:26.38 a message that is relatively new. 0:46:30 It is not an old message that I generated before that is now 0:46:34.214 being sent back to me again. That is what we are going to 0:46:38.214 use the timestamps here for. We also want to make sure that 0:46:42.357 the message is appropriate. And, by appropriate, 0:46:45.714 I am actually the intended recipient of this message and 0:46:49.642 the sender of this message is actually who the sender of this 0:46:53.928 message claimed to be. This message actually should be 0:46:57.714 applied right now. This message makes sense. 0:47:01.87 And then, finally, the third property we want is 0:47:05.386 something called forward secrecy. 0:47:07.78 What this says is essentially we should be able to change the 0:47:12.269 cryptographic keys that we are using at some point in the 0:47:16.458 future. Say, for example, 0:47:18.254 if our keys become compromised, we should be able to change the 0:47:22.892 keys and our method should continue to work. 0:47:26.109 This approach is pretty clearly true. 0:47:30 Alice could request a new key. And then you could imagine some 0:47:33.297 way in which Alice could create a new key and send that new key 0:47:36.648 to Bob and we can sort of start all over again. 0:47:39.135 The two properties that are really interesting are this 0:47:42.054 freshness property and this appropriateness property, 0:47:44.864 and they are the two properties that I want to talk about a 0:47:48 little bit more. And if you think about what the 0:47:50.54 kinds of attacks are that somebody could apply on this 0:47:53.405 thing, well, one attack they could apply would be an attack 0:47:56.54 on the sort of cryptographic transform itself. 0:48:00 A brute force attack where you try and factor the keys, 0:48:03.936 we said that is hard. Another attack that they could 0:48:07.653 apply would be a so-called replay attack. 0:48:10.569 There is a crypto attack and so-called replay attack. 0:48:14.359 This is that somebody might sort of reuse a message that 0:48:18.369 they saw transmitted over the network before. 0:48:21.576 Some Eve or Lucifer might overhear a message and then 0:48:25.366 might try and resend that message in order to get the 0:48:29.157 server to do something. Suppose there is a message that 0:48:33.501 tells the server to take some action in the outside world, 0:48:36.262 open a door, so you might save up a message 0:48:38.297 that somebody else had sent before that said open the door. 0:48:41.107 And then you might resend it in order to get the door open when 0:48:44.11 you wanted it to open. We want to prevent that. 0:48:46.339 And we are going to use timestamps to do this. 0:48:48.519 And getting the timestamps to work out is a little bit tricky. 0:48:51.474 If the sender and the receiver have perfectly synchronized 0:48:54.235 clocks, you might just say as long as the timestamp is within 0:48:57.141 the first few milliseconds that is fine. 0:49:00 But if that is not true then you need to do something a 0:49:03.103 little bit more sophisticated. There is a protocol called 0:49:06.321 Kerberos that works this out in detail. 0:49:08.505 And that, again, is described in the notes. 0:49:10.919 But the idea is just you put the timestamp in every message 0:49:14.252 that you send. And then when you sign the 0:49:16.551 message, when the person on the other end decodes that message, 0:49:20.114 they know that this message was generated as of a particular 0:49:23.505 time. But the third kind of attack we 0:49:25.574 might be worried about is a so-called impersonation attack. 0:49:30 And this is the attack that this protocol is susceptible to. 0:49:34.252 Does anybody see what the bug in this protocol is? 0:49:38 0:49:48 STUDENT: Charles does not authenticate that Alice is 0:49:53.186 Alice. Charles does not authenticate? 0:49:58.459 Well, Alice does send this key that says -- 0:50:05 Charles doesn't actually care because all Charles saw was a 0:50:09.252 message that said I want Alice's public key and B's public key. 0:50:13.797 There is nothing wrong with that message getting corrupted. 0:50:18.049 Alice might get something back, but if it comes back it has 0:50:22.302 been signed with Charles' private key and so she will know 0:50:26.48 exactly what it was that was in there. 0:50:30 STUDENT: Only Alice knows Charles private key? 0:50:33 I mean can't someone else impersonate Alice and do that 0:50:36.599 whole process and then be seen as Alice by Bob? 0:50:40 But notice that the key has been signed with Alice's private 0:50:44.719 key. This proposed key has been 0:50:47.119 signed with Alice's private key. STUDENT: Alice is sending 0:50:53.012 something encoded in Charles' private key, is that right? 0:51:00 No. Well, all this means is that 0:51:01.737 the thing that is encoded with Charles' private key, 0:51:04.594 Bob is going to be able to decode that thing using Charles' 0:51:07.844 public key. And he is going to have some 0:51:10.03 assurance, in fact, about what Alice's public key 0:51:12.719 is. I think that is OK. 0:51:13.952 Anybody else want to take a guess? 0:51:15.801 OK. Well, since we are out of time, 0:51:17.706 I will talk about this the first thing next time and will 0:51:20.844 show you what the problem with this protocol is. 0:51:23.478 I will see you on Monday. One announcement before you 0:51:27.498 guys go. I have a little gift for you 0:51:29.497 guys which is that we are going to cancel class next Wednesday 0:51:32.883 before the design project is due. 0:51:34.659 So there is no class next Wednesday, but there is class on 0:51:37.823 Monday so make sure you come here.