PROFESSOR: All right, guys, let's get started. So today, we're going to talk about network security. And in particular, we're going to talk about this paper on TCP/IP security by this guy Steve Bellovin, who used to be at AT&T and now is at Columbia. One interesting thing about this paper is it's actually a relatively old paper. It's more than 10 years old. And in fact, it's commentary on a paper that was 10 years before that. And many of you guys actually ask, why are we reading this if many of these problems have been solved in today's TCP protocol stacks?

So one interesting point-- so it's true that some of these problems that Steve describes in this paper have been solved since then. Some of them are still actually problems today. We'll sort of look at that and see what's going on. But you might actually wonder, why didn't people solve all these problems in the first place when they were designing TCP? What were they thinking?

And it's actually not clear. So what do you guys think? Why wasn't TCP designed to be secure with all these considerations up front? Yeah, any guesses? All right, anyone else? Yeah.

AUDIENCE: The internet was a much more trusting place back then.

PROFESSOR: Yeah, this was almost literally a quote from this guy's paper. Yeah, at the time-- the whole internet set of protocols was designed I guess about 40 years ago now. The requirements were totally different. It was to connect a bunch of relatively trusting sites that all knew each other by name.

And I think this is often the case in any system that becomes successful. The requirements change. So it used to be that this was a protocol for a small number of sites. Now it's the entire world. And you don't know all the people connected to the internet by name anymore. You can't call them up on the phone if they do something bad, et cetera.

So I think this is a story for many of the protocols we look at. And many of you guys have questions, like, what the hell were these guys thinking? This is so broken. But in fact, they were designing a totally different system. It got adopted.

Same for the web, like we were looking at in the last couple of weeks. It was designed for a very different goal. And it expanded. And you sort of have these growing pains you have to figure out how to make the protocol adapt to new requirements.

And another thing that somewhat suddenly happened is I think people also in the process gained a much greater appreciation for the kinds of problems you have to worry about in security. And it used to be the case that you didn't really understand all the things that you should worry about an attacker doing to your system.

And I think it's partly for this reason that it's sort of interesting to look at what happened to TCP security, what went wrong, how could we fix it, et cetera, to both figure out what kinds of problems you might want to avoid when designing your own protocols, and also what's the right mindset for thinking about these kinds of attacks. How do you figure out what an attacker might be able to do in your own protocol when you're designing it so you can avoid similar pitfalls?

All right, so with that preamble aside, let's actually start talking about what the paper is about. So how should we think about security in a network? So I guess we could try to start from first principles and try to figure out, what is our threat model? So what do we think the attacker is going to be able to do in our network?

Well, relatively straightforwardly, there's presumably being able to intercept packets, and probably being able to modify them. So if you send a packet over the network, it might be prudent to assume that some bad guy out there is going to see your packet and might be able to change it before it reaches the destination, might be able to drop it, and in fact might be able to inject packets of their own that you never sent with arbitrary contents.

And probably-- so this you can sort of come up with fairly straightforwardly by just thinking, well, if you don't trust the network, some bad guy is going to send arbitrary packets, see yours, modify them, et cetera. Somewhat more worryingly, as this paper talks about, the bad guy can also participate in your protocols. They have their own machine, right?

So the attacker has their own computer that they have full control over. So even if all the computers that you trust are reasonably maintained, they all behave correctly, the bad guy has his own computer that he can make it do whatever he wants. And in fact, he can participate in a protocol or distribute a system.

So if you have a routing protocol, which involves many people talking to each other, at some scale, it's probably going to be impractical to keep the bad guys out. If you're running a routing protocol with 10 participants, then maybe you can just call all them up and say, well, yeah, yeah, I know all you guys.

But at the scale of the internet today, it's unfeasible to have sort of direct knowledge of what everyone else or who everyone else in this protocol is. So probably some bad guy is going to be participating in your protocols or distributed systems. And it's important to design distributed systems that can nonetheless do something reasonable with that.

All right, so what are the implications of all these things? I guess we'll go down the list. So intercepting is-- it's on the whole easy to understand. Well, you shouldn't send any important data over the network if you expect a bad guy to intercept them, or at least not in clear text. Maybe you should encrypt your data.

So that seems relatively straightforward to sort of figure out. Although still you should sort of keep it in mind, of course, when designing protocols. Now, injecting packets turns out to lead to a much wider range of interesting problems that this paper talks about. And in particular, attackers can inject packets that can pretend to be from any other sender. Because the way this works in IP is that the IP packet itself has a header that contains the source of the packet and the destination.

And it's up to whoever creates the packet to fill in the right values for the source and destination. And no one checks that the source is necessarily the correct. There's some filtering going on these days. But it's sort of fairly spotty, and it's hard to rely on. So to a first approximation, an attacker could fill in any IP address as the source, and it will get to the destination correctly. And it's interesting to try to figure out what could an attacker do with such a capability of sending arbitrary packets.

Now, in the several weeks up to this, like in buffer overflows and web security, we looked at, to a large extent, implementation bugs, like, how could you exploit a buffer overflow? And interestingly, the author of this paper is actually not at all interested in implementation bugs. He's really interested in protocol errors or protocol mistakes.

So what's the big deal? Why is he down on implementation bugs, even though we spent several weeks looking at them? Why does it matter? Yeah.

AUDIENCE: Because we have to keep those bugs [INAUDIBLE].

PROFESSOR: Yeah, so this is the really big bummer about a bug in your protocol design. Because it's hard to change. So if you have an implementation bug, well, you had a memcpy or a print-out out of some sort that didn't check the range. OK, well, you had a range check, and it still works, and now it's also secure. So that's great.

But if you have some bug in the protocol specification, in how the protocol has to work, then fixing a bug is going to require fixing a protocol, which means potentially affecting all the systems that are out there speaking this protocol. So if we find some problem in TCP, it's potentially quite devastating. Because every machine that uses TCP is going to have to change. Because it's going to be hard to make it potentially backwards compatible.

We'll see exactly what these bugs are. But this is the real reason he's so excited about looking at protocol bugs. Because they're fairly fundamental to the TCP protocol that everyone agrees to speak.

So let's look at one of these guys. So the first example he points out has to do with how TCP sequence numbers work. So just to re-explain-- yeah, question.

AUDIENCE: I'm just curious. This is a tiny bit off topic. But let's say you do find a bug in TCP. How do you make the change to it? How do you tell all the computers in the world to change that?

PROFESSOR: Yeah, I think it's a huge problem. What if you find a bug in TCP? Well, it's unclear what to do. And I think the authors here struggle a lot with that. And in many ways, if you could redesign TCP, many of these bugs are relatively easy to fix if you knew what to look for ahead of time.

But because TCP is sort of relatively hard to fix or change, what ends up happening is that people or designers try to look for backwards compatible tweaks that either allow old implementations to coexist with the new implementation or to add some optional field that if it's there, then the communication is more secure in some way.

But it is a big problem. If it's some security issue that's deeply ingrained in TCP, then it's going to be a pretty humongous issue for everyone to just pack up and move onto a TCP version whatever, n plus 1. And you can look at IPv6 as one example of this not happening. We've known this problem was going to come up for like 15 years or 20 years.

IPv6 has been around for well over 10 years now. And it's just hard to convince people to move away from IPv4. It's good enough. It sort of works. It's a lot of overhead to move over. And no one else is speaking IPv6, so why should I start speaking this bizarre protocol that no one else is going to speak to me in?

So it's sort of moving along. But I think it takes a long time. And there's going to be really some motivation to migrate. And backwards compatibility helps a lot. Not good enough for, I guess, IPv6-- IPv6 has lots of backwards compatibility plans in it. You can talk to an IPv4 host from IPv6. So they try to engineer all this support. But still, it's hard to convince people to upgrade.

All right, but yeah, looking back at the TCP sequence numbers, we're going to look at actually two problems that have to do with how the TCP handshake works. So let's just spend a little bit of time working out what are the details of how a TCP connection gets initially established.

So there's actually three packets that have to get sent in order for a new TCP connection to be established. So our client generates a packet to connect to a server. And it says, well, here's my IP address, C, client. I'm sending this to the server.

And there's various fields. But the ones that are interesting for the purpose of this discussion is going to be a sequence number. So there's going to be a syn flag saying, I want to synchronize state and establish a new connection. And you include a client sequence number in the initial syn packet.

Then when the server receives this, the server is going to look and say, well, a client wants to connect to me, so I'll send a packet back to whatever this address is, whoever said they're trying to connect to me. So it'll send a packet from the server to the client and include its own synchronization number, SN server. And it'll acknowledge the client's number.

And finally, the client replies back, acknowledging the server synchronization number-- acknowledge SNS. And now the client can actually start sending data. So in order to send data, the client has to include some data in the packet, and also put in the sequence number of the client to indicate that this is actually sort of legitimate client data at the start of the connection. It's not some data from later on, for example, that just happens to arrive now because the server missed some initial parts of the data.

So generally, all these sequence numbers were meant for ensuring in order delivery of packets. So if the client sends two packets, the one that has the initial sequence number, that's the first chunk of data. And the one with the next sequence number is the next chunk of data.

But it turns out to also be useful for providing some security properties. Here's an example of these requirements changing. So initially, no one was thinking TCP provides any security properties. But then applications started using TCP and sort of relying on these TCP connections not being able to be broken by some arbitrary attacker, or an attacker not being able to inject data into your existing TCP connection. And all of a sudden, this mechanism that was initially meant for just packet ordering now gets used to guarantee some semblance of security for these connections.

So in this case, I guess the problem stems from what could a server assume about this TCP connection. So typically, the server assumes-- implicitly, you might imagine-- that this connection is established with the right client at this IP address C. It seems like a natural thing to assume.

Is there any basis for making this assumption? If a server gets this message saying, here's some data on this connection from a client to a server, and it has sequence number C, why might the server conclude that this was actually the real client sending this?

AUDIENCE: Because the sequence number is hard to guess.

PROFESSOR: Right, so that's sort of the implicit thing going on, that it has to have the right sequence number C here. And in order for this connection to get established, the client must have acknowledged the server sequence number S here. And the server sequence number S was only sent by the server to the intended client IP address. Yeah.

AUDIENCE: How many bits are available for the sequence number?

PROFESSOR: So sequence numbers in TCP are 32 bits long. That's not entirely easy to guess. If it was really a random 32 bit number, it would be hard to just guess. And you'd probably waste a lot of bandwidth trying to guess this. Yeah, question.

AUDIENCE: The data frequency number is higher than the initial sequence number?

PROFESSOR: Yeah, so basically, these things get incremented. So every time you send a syn, that counts as one byte against your sequence number. So this is SNC. I think actually what happens is this is SNC plus 1. And then it goes on from there.

So if you send 5 bytes, then the next one is SNC initial plus 6. So this just counts the bytes that you're sending. SYNs count as 1 byte each. Make sense? Other questions about this?

All right, so typically, or at least the way the TCP specification recommended that people choose these sequence numbers, was to increment them at some roughly fixed rate. So the initial RFCs suggested that you increment these things at something like 250,000 units, plus 250,000, per second.

And the reason that it wasn't entirely random is that these sequence numbers are actually used to prevent out of order packets, or packets from previous connections, from interfering with new connections. So if every time you established a new connection you chose a completely random sequence number, then there's some chance if you establish lots of connections over and over that some packet from a previous connection is going to have a similar enough sequence number to your new connection and is going to be accepted as a valid piece of data on that new connection.

So this is something that the TCP designers worried a lot about-- these out of order packets or delayed packets. So as a result, they really wanted these sequence numbers to progress in a roughly monotonic matter over time, even across connections. If I opened one connection, it might have the same source and destination, port numbers, IP addresses, et cetera. But because I established this connection now instead of earlier, packets from earlier hopefully aren't going to match up with the sequence numbers I have for my new connection. So this was a mechanism to prevent confusion across repeated connection establishments. Yeah.

AUDIENCE: So if you don't know exactly how much your other grid that you're talking to is going to improve the sequencing pack, how do you know that the packet you're getting is the next packet if there wasn't [INAUDIBLE] immediate packet that you--

PROFESSOR: So typically you'll remember the last packet that you received. And if the next sequence number is exactly that, then this is the next packet in sequence. So for example, here, the server knows that I've seen exactly SNC plus 1 worth of data. If the next packet has sequence number SNC plus 1, that's the next one.

AUDIENCE: So you're saying that when you establish a sequence number, then even after that you're committing it--

PROFESSOR: Well, absolutely, yeah, yeah. So these sequence numbers, initially when you establish it, they get picked according to some plan. We'll talk about that plan. You can sort of think they might be random. But over time, they have to have some flow for initial sequence numbers for connection.

But within a connection, once they're established, that's it. They're fixed. And they just tick along as the data gets sent on the connection, exactly. Make sense? All right, so there were some plans suggested for how to manage these sequence numbers. And it was actually a reasonable plan for avoiding duplicate packets in the network causing trouble.

But the problem, of course, showed up that attackers were able to sort of guess these sequence numbers. Because there wasn't a lot of randomness being chosen. So the way that the host machine would choose these sequence numbers is they have just a running counter in memory. Every second they bump it by 250,000. And every time a new connection comes in, they also bump it by some constant like 64k or 128k. I forget the exact number. So this was relatively easy to guess, as you can tell. You send them their connection request, and you see what sequence number comes back. And then you know the next one is going to be 64k higher than that. So there wasn't a huge amount of randomness in this protocol.

So we can just sketch out what this looks like. So if I'm an attacker that wants to connect to a server but pretend to be from a particular IP address, then what I might do is send a request to the server, very much like the first step there, include some initial sequence number that I choose. At this point, any sequence number is just as good, because the server shouldn't have any assumptions about what the client's sequence number is.

Now, what does the server do? The server gets the same packet as before. So it performs the same way as before. It sends a packet back to the client with some server sequence number and acknowledges SNC. And now the attacker, if the attacker wants to establish a connection, needs to somehow synthesize a packet that looks exactly like the third packet over there. So it needs to send a packet from the client to the server.

That's easy enough. You just fill in these values in the header. But you have to acknowledge this server sequence number SNS. And this is where sort of the problems start. If the SNS value is relatively easy to guess, then the attacker is good to go. And now the server thinks they have an established connection with a client coming from this IP address.

And now an attacker could inject data into this connection just as before. They just synthesize a packet that looks like this, has the data, and it has the client sequence number that in fact the adversary chose. Maybe it's plus 1 here. But it all hinges on being able to guess this particular server supplied sequence number. All right, does this make sense? Yeah.

AUDIENCE: What's the reason that the server sequence number isn't completely random?

PROFESSOR: So there's two reasons. One, as I was describing earlier, the server wants to make sure that packets from different connections over time don't get confused for one another. So if you establish a connection from one source port to another destination port, and then you close the connection and establish another one of the same source and destination port, you want to make sure the packets from one connection don't appear to be valid in another connection.

AUDIENCE: So the server sequence number is incremented for every one of their packets?

PROFESSOR: Well, so the sequence numbers within a connection, as I was describing, get bumped with all the data in a connection. But there's also the question of, how do you choose the initial sequence number here?

And that gets bumped every time a new connection is established. So the hope is that by the time it wraps around 2 to the 32 and comes back, there's been enough time so that old packets in the network have actually been dropped and will not appear as duplicates anymore. So that's the reason why you don't just choose random points, or they didn't initially choose random points. Yeah.

AUDIENCE: So this is a problem between connections, for a connection between the same guide, the same client, the same server, the same source port, the same destination. And we're worried about old packets--

PROFESSOR: So this is what the original, yeah, TCP designers were worried about, which is why they prescribed this way of picking these initial sequence numbers.

AUDIENCE: If you have different new connections, you could differentiate.

PROFESSOR: That's right, yeah.

AUDIENCE: So then I don't see why the incrementing stuff and not just take randomly.

PROFESSOR: So I think the reason they don't pick randomly is that if you did pick randomly, and you established, I don't know, 1,000 connections within a short amount of time from the same source to the same destination, then, well, every one of them is some random value of module

2 to the 32. And now there's a nontrivial chance that some packet from one connection will be delayed in the network, and eventually show up again, and will get confused for a packet from another connection. This is just sort of nothing to do with security. This is just their design consideration initially for reliable delivery.

AUDIENCE: [INAUDIBLE] some other client to the server, right?

PROFESSOR: Sorry?

AUDIENCE: This is [INAUDIBLE] some other client?

PROFESSOR: That's right, yeah. So we haven't actually said why this is interesting at all for the attacker to do. Why bother? You could just go from his old IP address, right?

AUDIENCE: So what happens for the server [INAUDIBLE]?

PROFESSOR: Yes, this is actually an interesting question. What happens here? So this packet doesn't just get dropped. It actually goes to this computer. And what happens?

AUDIENCE: [INAUDIBLE], they just mentioned you try and do it like they would try and do it when the other computer was updating or rebooting or off, or something.

PROFESSOR: Right, certainly they felt, oh, that computer is offline. The packet will just get dropped, and you don't have to worry about it too much. If a computer is actually listening on that IP address, then in the TCP protocol, you're supposed to send a reset packet resetting the connection. Because this is not a connection that computer C knows about.

And in TCP, this is presumed to be because, oh, this is some old packet that I requested long ago, but I've since forgotten about it. So the machine C here might send a packet to the server saying, I want a reset. I actually forget exactly which sequence number goes in there. But the client C here knows all the sequence numbers and send any sequence number as necessary and reset this connection.

So if this computer C is going to do this, then it might interfere with your plan to establish a connection. Because when S gets this packet, it says, oh, sure, if you don't want it, I'll reset your connection.

There's some implementation-ish bugs that you might exploit, or at least the author talks about, and an potentially exploiting, that would prevent client C from responding. So for example, if you flood C with lots of packets, it's an easy way to get him to drop this one. It turns out there are other more interesting bugs that don't require flooding C with lots of packets that still get C to drop this packet, or at least it used to on some implementations on TCP stacks. Yeah.

AUDIENCE: Presumably, most firewalls would also [INAUDIBLE].

PROFESSOR: This one?

AUDIENCE: No, the SYN.

PROFESSOR: This one.

AUDIENCE: That came into a client, and a client didn't originally send a SYN to that server. And the firewall is going to drop it.

PROFESSOR: It depends, yeah. So certainly if you have a very sophisticated stateful firewall that keeps track of all existing connections, or for example if you have a NAT, then this might happen. On the other hand, a NAT might actually send the RST on behalf of the client.

So it's not clear. I think this is not as common. So for example, on a Comcast network, I certainly don't have anyone intercepting these packets and maintaining state for me and sending RSTs on my behalf or anything like that. Yeah.

AUDIENCE: So why can't the server have independent sequence numbers for each possible source?

PROFESSOR: Right, so this is in fact what TCP stacks do today. This is one example of how you fix this problem in a backwards compatible manner. So we'll get to exactly the formulation of how you arrange this. But yeah, it turns out that if you look at this carefully, as you're doing, you don't need to have this initial sequence number be global. You just scope it to every source/destination pair. And then you have all the duplicate avoidance properties we had before, and you have some security as well.

So just to sort of write this out on the board of how the attacker is getting this initial sequence number, the attacker would probably just send a connection from its own IP address to the server saying, I want to establish a new connection, and the server would send a response back to the attacker containing its own sequence number S. And if the SNS for this connection and the SNS for this connection are related, then this is a problem.

But you're saying, let's make them not related. Because this is from a different address. Then this is not a problem anymore. You can't guess what this SNS is going to be based on this SNS for a different connection. Yeah.

AUDIENCE: So you still have a collision problem, because you could engage the 32 bits by the addresses of your peers. So you have a lot of ports for each one of these. So you still have conflicting sequence numbers for all of these connections that you're getting, right?

PROFESSOR: So these sequence numbers are specific, as it turns out, to an IP address and a port number source/destination duple. So if it's different ports, then they don't interfere with each other at all.

AUDIENCE: Oh, because you're using the port--

PROFESSOR: That's right, yeah, you also use the port in this as well.

AUDIENCE: Because I thought those ports--

PROFESSOR: Yeah, so the ports are sort of below the sequence numbers in some way of thinking about it. Question?

AUDIENCE: If the sequence numbers are global, then doesn't the attacker [INAUDIBLE]?

PROFESSOR: Yeah, good point. So in fact, if the server increments the sequence number by, I don't know, 64k I think it is, or it was, for every connection, then, well, you connect. And then maybe five other people connect. And then you have to do this attack.

So to some extent, you're right, this is a little troublesome. On the other hand, you could probably arrange it for your packet here to be delivered just before this packet. So if you send these guys back to back, then there's a good chance they'll arrive at the server back to back.

The server will get this one, respond with this sequence number. It'll get the next one, this one, respond with the sequence number right afterwards. And then you know exactly what to put in this third packet in your sequence.

So I think this is not a foolproof method of connecting to a server. There's some guessing involved. But if you carefully arrange your packets right, then it's quite easy to make the right guess. Or maybe you try several times, and you'll get lucky. Yeah.

AUDIENCE: So even if it's totally random, and you have to guess it, there are only like 4 billion possibilities. It's not a huge number, right? I feel like in the course of a year, you should be able to probably get through.

PROFESSOR: Right, yeah, so you're absolutely right. You shouldn't really be relying on TCP to provide security very strongly. Because you're right, it's only 4 billion guesses. And you can probably send that many packets certainly within a day if you have a fast enough connection.

So it's sort of an interesting argument we're having here in the sense that at some level, TCP is hopefully insecure. Because it's only 32 bits. There's no way we could make it secure. But I think many applications rely on it enough that not providing any security at all is so much of a nuisance that it really becomes a problem.

But you're absolutely right. In practice, you do want to do some sort of encryption on top of this that will provide stronger guarantees that no one tampered with your data, but where the keys are more than 32 bits long. It still turns out to be useful to prevent people from tampering with TCP connections in most cases.

All right, other questions? All right, so let's see what actually goes wrong. Why is it a bad thing if people are able to spoof TCP connections from arbitrary addresses? So one reason why this is bad is if there is any kind of IP-based authorization. So if some server decides whether an operation is going to be allowed or not based on the IP address it comes from, then this is potentially going to be a problem for an attacker who spoofed connections from an arbitrary source address.

So one example where this was a problem-- and it largely isn't anymore-- is this family of r commands, things like rlogin. So it used to be the case that you could run something like rlogin into a machine, let's say athena.dialup.mit.edu. And if your connection was coming from a host at MIT, then this rlogin command would succeed if you say, oh yeah, I'm user Alice on this machine. Let me log in as user Alice onto this other machine. And it'll just trust that all the machines at mit.edu are trustworthy to make these statements.

I should say I think dial-up never actually had this problem. It was using Cerberus from the very beginning. But other systems certainly did have such problems. And this is an example of using the IP address where the connection is coming from some sort of authentication mechanism for whether the caller or the client is trustworthy or not. So this certainly used to be a problem, isn't a problem anymore. So relying on IP seems like such a clearly bad plan.

Yet, this actually is still the case. So rlogin is gone. It was recently replaced by SSH now, which is good. On the other hand, there are still many other examples of protocols that rely on IP-based authentication.

One of them is SMTP. So when you send email, you use SMTP to talk to some mail server to send a message. And to prevent spam, many SMTP servers will only accept incoming messages from a particular source IP address. So for example, Comcast's mail server will only accept mail from Comcast IP addresses. Same for MIT mail servers-- will only accept mail from MIT IP addresses. Or there was at least one server that ISNT runs that has this property.

So this is the case where it's still using IP-based authentication. Here it's not so bad. Worst case, you'll send some piece of spam through the mail server. So that's probably why they're still using it, whereas things that allow you to log into an arbitrary account stopped using IP-based authentication. So does this make sense, why this is a bad plan?

And just to double check, suppose that some server was using rlogin. What would you do to attack it? What bad thing would happen? Suggestions? Yeah.

AUDIENCE: Just getting into your computer, and then make a user that you want to log into, and then you get into the network.

PROFESSOR: Yeah, so basically you get your computer. You synthesize this data to look like a legitimate set of rlogin commands that say, log in as this user and run this command in my Unix shell there. You sort of synthesize this data and you mount this whole attack and send this data as if a legitimate user was interacting with an rlogin client, and then you're good to go.

OK, so this is one reason why you probably don't want your TCP sequence numbers to be so guessable. Another problem is these reset attacks. So much like we were able to send a SYN packet, if you know someone's sequence number, you could also send a reset packet. We sort of briefly talked about it here as the legitimate client potentially sending a reset to reset the fake connection that the attacker is establishing.

But in a similar vain, the adversary could try to send reset packets for an existing connection if there's some way that the adversary knows what your sequence number is on that connection. So this is actually not clear if this is such a big problem or or.

At some level, maybe you should be assuming that all your TCP connections could be broken at any time anyway. It's not like the network is reliable. So maybe you should be expecting your connections to drop.

But one place where this turned out to be particularly not a good assumption to make is in the case of routers talking to one another. So if you have multiple routers that speak some routing protocol, then they're connected, of course, by some physical links. But over some physical links, they actually speak some network protocol. And that network protocol runs over TCP. So there's actually some TCP session running over each of these physical links that the routers use to exchange routing information.

So this is certainly the case for this protocol called BGP we'll talk about a bit more in a second. And BGP uses the fact that the TCP connection is alive to also infer that the link is alive. So if the TCP connection breaks, then the routers assume the link broke. And they recompute all their routing tables.

So if an adversary wants to mount some sort of a denial of service attack here, they could try to guess the sequence numbers of these routers and reset these sessions. So if the TCP session between two routers goes down, both routers are like, oh, this link is dead. We have to recompute all the routing tables, and the routes change. And then you might shoot down another link, and so on.

So this is a bit of a worrisome attack, not because it violates someone's secrecy, et cetera, or at least not directly, but more because it really causes a lot of availability problems for other users in the system. Yeah.

AUDIENCE: So if you're an attacker, and you wanted to target one particular user, could you just keep sending connection requests to a server on behalf of his IP and make him keep dropping his connections to the servers and so you just [INAUDIBLE]?

PROFESSOR: Well, so it requires you guessing. So you're saying, suppose I'm using Gmail, and you want to stop me from learning something in Gmail, so just send packets to my machine pretending to be from Gmail. Well, you have to guess the right source and destination port numbers.

The destination port number is probably 443, because I'm using HTTPS. But the source port number is going to be some random 16-bit thing. And it's also going to be the case that probably the sequence numbers are going to be different. So unless you guess a sequence number that's within my TCP window, which is in order of probably tens of kilobytes, you're also going to be not successful in that regard.

So you have to guess a fair amount of stuff. There's no sort of oracle access. You can't just query the server and say, well, what is that guy's sequence number? So that's the reason why that doesn't work out as well.

So again, many of these issues were fixed, including this RST-based thing, especially for BGP routers. There was actually two sort of amusing fixes. One really shows you how you can carefully exploit existing things or take advantage of them to fix particular problems. Here, the insight is that these routers only want to talk to each other, not to someone else over the network. And as a result, if the packet is coming not from the immediate router next across the link, but from someone else, I want to drop this packet all together.

And what the designers of these writing protocols realized is that there's this wonderful field in a packet called time to live. It's an 8-bit field that gets decremented by every router to make sure that packets don't go into an infinite loop. So the highest this TTL value could ever be is 255. And then it'll get decremented from there.

So what these writing protocols do-- it's sort of a clever hack-- is they reject any packet with a TTL value that's not 255. Because if a packet has a value of 255, it must have come from the router just on the other side of this link. And if the an adversary tries to inject any packet to tamper with this existing BGP connection, it'll have a TTL value less than 255, because it'll be decremented by some other routers along the path, including this one.

And then it'll just get rejected by the recipient. So this is one example of a clever combination of techniques that's backwards compatible and solves this very specific problem. Yeah.

AUDIENCE: Doesn't the bottom right router also send something with a TTL of 255?

PROFESSOR: Yeah, so these routers are actually-- this is a physical router. And it knows these are separate links. So it looks at the TTL and which link it came on. So if a packet came in on this link, it will not accept it for this TCP connection.

But you're right. For the most part, these routers trust their immediate neighbors. It need not necessarily be the case. But if you keep seeing this problem, and you know you've implemented this hack, then it must be one of your neighbors. You're going to look. TCP dumped these interfaces. Why are you sending me these reset packets?

This problem is not as big. You can manage it by some Auto Pan mechanism. Make sense? All right, there are other fixes for BGP where they implemented some form of header authentication, MD5 header authentication as well. But they're really targeting this particular application where this reset attack is particularly bad.

This is still a problem today. If there's some long-lived connection out there that I really want to shoot down, I just have to send some large number of RST packets, probably on the order of hundreds of thousands or so, but probably not exactly 4 billion. Because the servers are actually somewhat lax in terms of which sequence number they accept for a reset.

It can be any packet within a certain window. And in that case, I could probably, or any attacker, reset an existing connection with a modest but not a huge amount of effort. That's still a problem. And people haven't really found any great solution for that.

All right, and I guess the sort of last bad thing that happens because these sequence numbers are somewhat predictable is just data injection into existing connections. So suppose there is some protocol like rlogin, but maybe rlogin doesn't-- suppose we have some hypothetical protocol that's kind of like rlogin, but actually it doesn't do IP-based authentication. You have to type in your password to log in, all this great stuff.

The problem is once you've typed your password, maybe your TCP connection is just established and can accept arbitrary data. So wait for one of you guys to log into a machine, type in your password. I don't know what that password is. But once you've established TCP connection, I'll just try to guess your sequence number and inject some data into your existing connection. So if I can guess your sequence numbers correctly, then this allows me to make it pretend like you've typed some command after you authenticated correctly with your password.

So this all sort of suggests that you really don't want to rely on these 32-bit sequence numbers for providing security. But let's actually see what modern TCP stacks actually do to try to mitigate this problem. So as we were sort of discussing, I guess one approach that we'll look at in the next two lectures is how to implement some security at the application level. So we'll use cryptography to authenticate and encrypt and sign and verify messages at the application level without really involving TCP so much.

But there are some existing applications that would benefit from making this slightly better, at least not make it so easy to exploit these problems. And the way that I guess people do this in practice today-- for example Linux and Windows-- is they implement the suggestion that John gave earlier, that we maintain different initial sequence numbers for every source destination pair.

So what most TCP SYN implementations do is they still compute this initial sequence number as we were computing before. So this is the old style ISN, let's say. And in order to actually generate the actual ISN for any particular connection, we're going to add a random 32-bit offset. So we're going to include some sort of a function. Think of it like as like a hash function like SHA-1 or something maybe better.

And this is going to be a function of the source IP, the source port number, the destination IP address, destination port, and some sort of a secret key that only the server knows in this case. So this has the nice property that within any particular connection, as identified by a source and destination IP port pair, it still preserves all these nice properties of this old style sequence number algorithm had.

But if you have connections from different source/destination tuples, then there's nothing you can learn about the exact value of another connection tuple's sequence number. And in fact, you'll have to guess this key in order to infer that value. And hopefully the server, presumably the OS kernel, stores this key somewhere in its memory and doesn't give it out to anyone else.

So this is how pretty much most TCP stacks deal with this particular problem today to the extent allowed by the total 32-bit sequence number. It's not great, but sort of works. Yeah.

AUDIENCE: Could you repeat that again? Is the key unique to--

PROFESSOR: So when my machine boots up, or when any machine boots up, it generates a random key. Every time you reboot it it generates a new key. And this means that every time that for a particular source/destination pair, the sequence numbers advance at the same rate as controlled by this.

So for a given source/destination pair, this thing is fixed. So you observe your sequence numbers evolving according to your initial sequence numbers for new connections evolving according to a particular algorithm. So that still provides all these defences against old packets from previous connections being injected into new connections, just like packet reordering problems.

So that still works. And that's the only real thing for which we needed this sequence number choosing algorithms to prevent these duplicate packets from causing problems. However, the thing that we were exploiting before, which is that if you get the sequence number for one connection from A to S, then from that you can infer the sequence number for a different connection.

That's now gone. Because every connection has a different offset in this 32-bit space as implemented by its F function. So this completely decouples the initial sequence numbers seen by every connection. Yeah.

AUDIENCE: What's the point in including the key?

PROFESSOR: Well, if you don't include the key, then I can connect to you. I'll compute the same function F. I'll subtract it out. I'll get this. I'll compute this function F for the connection I actually want to fake. And I'll guess what the initial sequence number for that one is going to be.

AUDIENCE: So can you-- because machines now restart infrequently, can you still [INAUDIBLE] by reversing--

PROFESSOR: I think typically this function F is something like a cryptographically secure hash function, which has a semi-proved property that it's very difficult. It's cryptographically hard to invert it. So even if you were given the literal inputs and outputs of this hash function except for this key part, it would be very hard for you guess what this key is cryptographically, even in an isolated setting.

So hopefully this will be at least as hard in this setting as well. We'll talk a little bit more about what these functions F are a bit later on and how you to use them correctly. Make sense? Other questions of this problem and solution?

All right, so in fact, this was mostly sort of an example of these TCP sequence number attacks that aren't as relevant anymore. Because every operating system basically implements this plan these days. So it's hard to infer what someone's sequence number is going to be.

On the other hand, people keep making the same mistakes. So even after this was implemented for TCP, there was this other protocol called DNS that is hugely vulnerable to similar attacks. And the reason is that DNS actually runs over UDP.

So UDP is a stateless protocol where you actually don't do any connection establishment where you exchange sequence numbers. In UDP, you simply send a request from your source address to the server. And the server figures out what the reply should be and sends it back to whatever source address appeared in the packet.

So it's a single round trip, so there's no time to exchange sequence numbers and to establish that, oh, yeah, you're actually talking to the right guy. So with DNS, as a result, for a while, it was quite easy to fake responses from a DNS server.

So how would a query look like in DNS, typically? Well, you send some queries-- so suppose a client sends a packet from client to some DNS server that knows the DNS server's IP address ahead of time, maybe preconfigured somewhere, say, well, here's my query. Maybe I'm looking for mit.edu. And that's basically it.

And the server's destination port number is always 53 for DNS. And the clients used to also run on the same port number for ease of use or something. So you send this packet from the client on this port to the server on this port. Here's the query. And the server eventually sends back a reply saying, mit.edu has a particular IP address, 18.9 dot something.

The problem is that some adversary could easily send a similar response packet pretending to be from the server. And there's not a whole lot of randomness here. So if I know that you're trying to connect to mit.edu, I'll just send a lot of packets like this to your machine.

I know exactly what DNS server you're going to query. I know exactly what your IP address is. I know the port numbers. I know what you're querying for. I can just supply my own IP address here.

And if my packet gets there after you send this but before you get the real response, your client machine is going to use my packet. So this is another example where insufficient randomness in this protocol makes it very easy to inject responses or inject packets in general.

And this is actually in some ways even worse than the previous attack. Because here you could convince a client to connect to another IP address all together. And it'll probably cache this result, because DNS involves caching. Maybe you can supply a very long time to live in this response saying, this is valid for years. And then your client, again till it reboots, is going to keep using this IP address for mit.edu. Yeah.

AUDIENCE: Could you fix this by having the client include some random value in the query, and the server customer exactly?

PROFESSOR: That's right, yeah, so this is typically what people have done now. The problem, as we were sort of talking about earlier, is backward compatibility. It's very hard to change the DNS server software that everyone runs.

So you basically have to figure out, where can you inject randomness? And people have figured out two places. It's not great. But basically there's a source port number, which is 16 bits of randomness. So if you can choose the source port number randomly, then you get 16 bits. And there's also a query ID inside of the packet, which is also 16 bits. And the server does echo back the query ID.

So combining these two things together, most resolvers these days get 32 bits of randomness out of this protocol. And it, again, makes it noticeably harder, but still not cryptographically perfect, to fake this kind of response and have it be accepted by the client. But these problems keep coming up, unfortunately. So even though it was well understood for TCP, some people I guess suggested that this might be a problem. But it wasn't actually fixed until only a few years ago. Make sense?

All right, so I guess maybe as an aside, there are solutions to this DNS problem as well by enforcing security for DNS at the application level. So instead of relying on these randomness properties of small numbers of bits in the packet, you could try to use encryption in the DNS protocols. So protocols like DNS SEC that the paper briefly talks about try to do this. So instead of relying on any network level security properties, they require that all DNS names have signatures attached to them.

That seems like a sensible plan. But it turns out that working out the details is actually quite difficult. So one example of a problem that showed up is name and origin. Because in DNS, you want to get responses. Well, this name has that IP address.

Or you could get a response saying, no, so sorry, this name doesn't exist. So you want to sign the it doesn't exist response as well. Because otherwise, that adversary could send back a doesn't exist response and pretend that a name doesn't exist, even though it does. So how do you sign responses that certain names don't exist ahead of time? I guess one possibility is you could give your DNS server the key that signs all your records.

That seems like a bad plan. Because then someone who compromises your DNS server could walk away with this key. So instead, the model the DNS SEC operates under is that you sign all your names in your domain ahead of time, and you give the signed blob to your DNS server. And the DNS server can then respond to any queries. But even if it's compromised, there's not much else that that attacker can do. All these things are signed, and the key is not to be found on the DNS server itself.

So the DNS SEC protocol had this clever mechanism called NSEC for signing nonexistent records. And the way you would do this is by signing gaps in the namespace. So an NSEC record might say, well, there's a name called foo.mit.edu, and the next name alphabetically is maybe goo.mit.edu.

And there's nothing alphabetical in between these two names. So if you query for a name between these two names alphabetically sorted, then the server could send back this signed message saying, oh, there's nothing between these two names. You can safely return, doesn't exist.

But then this allows some attacker to completely enumerate your domain name. You can just ask for some domain name and find this record and say, oh, yeah, great. So these two things exist. Let me query for gooa.mit.edu. That'll give me a response saying, what's the next name in your domain, et cetera.

So it's actually a little bit hard to come up with the right protocol that both preserves all the nice properties of DNS and prevents name enumeration and other problems. There's actually a nice thing now called NSEC3 that tries to solve this problem partially-- sort of works, sort of not. We'll see, I guess, what gets it [INAUDIBLE]. Yeah.

AUDIENCE: Is there any kind of signing of nonexistent top level domains?

PROFESSOR: Yeah, I think actually yeah. The dot domain is just another domain. And they similarly have this mechanism implemented as well. So actually dot and dot com now implement DNS SEC, and there's all these records there that say, well, .in is a domain name that exists, and dot something else exists, and there's nothing in between. So there's all these things.

AUDIENCE: So other than denial of service, why do we care so much about repeating domain names within mit.edu?

PROFESSOR: Well, probably we don't. Actually, there's a text file in AFS that lists all these domain names at MIT anyway. But I think in general, some companies feel a little uneasy about revealing this.

They often have internal names that sit in DNS that should never be exposed to the outside. I think it's actually this fuzzy area where it was never really formalized what guarantees DNS was providing to you or was not. And people started assuming things like, well, if we stick some name, and it's not really publicized anywhere, then it's probably secure here.

I think this is another place where this system doesn't have a clear spec in terms of what it has and doesn't have to provide. And when you make some changes like this, then people say, oh, yeah, I was sort of relying on that. Yeah.

AUDIENCE: [INAUDIBLE] replay attack where you could send in bold gap signature?

PROFESSOR: Yeah, there's actually time outs on these things. So when you sign this, you actually sign and say, I'm signing that this set of names is valid for, I don't know, a week. And then the clients, if they have a synchronized clock, they can reject old signed messages. Make sense?

All right, so this is on the TCP SYN guessing attacks. Another interesting problem that also comes up in the TCP case is a denial of service attack that exploits the fact that the server has to store some state. So if you look at this handshake that we had on the board before, we'll see that when a client establishes a connection to the server, the server has to actually remember the sequence number SNC. So the server has to maintain some data structure on the side that says, for this connection, here's the sequence number.

And it's going to say, well, my connection from C to S has the sequence number SNC. And the reason the server has to store this table is because the server needs to figure out what SNC value to accept here later. Does this make sense?

AUDIENCE: [INAUDIBLE] SNS?

PROFESSOR: Yeah, the server also needs SNS I guess, yeah. But it turns out that-- well, yeah, you're right. And the problem is that-- actually, yeah, you're right. SNS is actually much more important. Sorry, yeah. [INAUDIBLE] SNS is actually much more important. Because SNS is how you know that you're talking to the right guy.

The problem is that there's no real bound on the size of this table. So you might get packets from some machine. You don't even know who sent it. You just get a packet that looks like this with a source address that claims to be C.

And in order to potentially accept a connection later from this IP address, you have to create this table entry. And these table entries are somewhat long lived. Because maybe someone is connecting to you from a really far away place. There's lots of packet loss. It might be not for maybe a minute until someone finishes this TCP handshake in the worst case.

So you have to store this state in your TCP stack for a relatively long time. And there's no way to guess whether this is a valid connection or not. So one denial of service attack that people discovered against most TCP stacks is to simply send lots of packets like this.

So if I'm an attacker, then I'll just send lots of SYN packets to a particular server and get it to fill up its table. And the problem is that in the best case, maybe the attacker just always uses the same source IP address.

In that case, you can just say, well, every client machine is allowed two entries in my table, or something like this. And then the attacker can use up two table entries but not much more. The problem, of course, is that the attacker can fake these client IP addresses, make them look random. And then for the server, it's going to be very difficult to distinguish whether this is an attacker trying to connect to me or some client I've never heard of before.

So if you're some website that's supposed to accept connections from anywhere in the world, this is going to be a big problem. Because either you deny access to everyone, or you have a store state for all these mostly fake connection attempts. Does that make sense?

So this is a bit of a problem for TCP, and in fact for most protocols that allow some sort of connection initiation, and the server has to store state. So there's some fixes. We'll talk about in a second what workaround TCP implements to try to deal with this problem. This is called SYN flooding in TCP.

But in general, this is a problem that's worth knowing about and trying to avoid in any protocol you design on top as well. So you want to make sure that the server doesn't have to keep state until it can actually authenticate and identify, who is the client?

Because by that time, if you've identified who the client is, you've authenticated them somehow, then you can actually make a decision, well, every client is allowed to only connect once, or something. And then I'm not going to keep more state.

Here, the problem is you're guaranteeing that you're storing state before you have any idea who it is that is connecting to you. So let's look at how you can actually solve this SYN flooding attack where the server accumulates lots of state.

So of course, if you could change TCP again, you could fix this pretty easily by using cryptography or something or changing exactly who's responsible for storing what state. The problem is we have TCP as is. And could we fix this problem without changing the TCP wire protocol? So this is, again, an exercise in trying to figure out, well, what exactly tricks we could play or exactly what assumptions we could relax and still stick to the TCP header format and other things.

And the trick is to in fact figure out a clever way to make the server stateless without having to-- so the server isn't going to have to keep this table around in memory. And the way we're going to do this is by carefully choosing SMS. Instead of using this formula we were looking at before, where we were to add this function, we're instead going to choose this sequence number in a different way.

And I'll give you exactly the formula. And then we'll talk about why this is actually interesting and what nice properties it has. So if the server detects that it's under this kind of attack, it's going to switch into this mode where it chooses SNS using this formula of applying basically the same or similar kind of function F we saw before. And what it's going to apply it to is the source IP, destination IP, the same things as before, source port, destination port, and also timestamp, and also a key in here as well.

And we're going to concatenate it with a timestamp as well. So this timestamp is going to be fairly coarse grained. It's going to go in order of minutes. So every minute, the timestamp ticks off by one. It's a very coarse grained time.

And there's probably some split between this part of the header and this part of the header. This timestamp doesn't need a whole lot of bits. So I forget exactly what this protocol does in real machines. But you could easily imagine maybe using 8 bits. For the timestamp, I'm going to be using 24 bits for this chunk of the sequence number.

All right, so why is this a good plan? What's going on here? Why this weird formula? So I think you have to remember, one was the property that we were trying to achieve of the sequence number. So there's two things going on. One is there's this defense against duplicated packets that we were trying to achieve by-- maybe the formula is still here. Nope-- oh, yeah, yeah, here.

Right, so just to compare these guys-- so when we're not under attack, we were previously maintaining this old style sequence number scheme to prevent duplicate packets from previous connections, all this good stuff. It turns out people couldn't figure out a way to defend against these kinds of SYN flooding attacks without giving up on this property, so basically saying, well, here's one plan that works well in some situations. Here's a different plan where we'll give up on that ISN old style component.

And instead, we'll focus on just ensuring that if someone presents us this sequence number S in response to a packet, like here, then we know it must've been the right client. So remember that in order to prevent IP spoofing attacks, we sort of rely on this SNS value. So if the server sends this SNS value to some client, then hopefully only that client can send us back the correct SNS value, finish establishing the connection.

And this is why you had to store it in this table over here. Because otherwise, how do you know if this is a real response or a fake response? And the reason for using this function F here is that now we can maybe not store this table in memory. And instead, when a connection attempt arrives here, we're going to compute SNS according to this formula over here and just send it back to whatever client pretends to have connected to us.

And then we'll forget all about this connection. And then if this third packet eventually comes through, and its SNS value here matches what we would expect to see, then we'll say, oh yeah, this must've been someone got our response from step two and finally sent it back to us.

And now we finally commit after step three to storing a real entry for this TCP connection in memory. So this is a way to sort of defer the storage of this state at the server by requiring the server, the client, to echo back this exact value. And by constructing it in this careful way, we can actually check whether the client just made up this value, or if it's the real thing we're expecting. Does that make sense?

AUDIENCE: [INAUDIBLE] SNC [INAUDIBLE]?

PROFESSOR: Yeah, so SNC now, we basically don't store it. It's maybe not great. But so it is. So in fact, I guess what really happens is in-- I didn't show it here. But there's probably going to be sort of a null data field here that says this packet has no data. But it still includes the sequence number SNC just because there's a field for it.

So this is how the server can reconstruct what this SNC value is. Because the client is going to include it in this packet anyway. It wasn't relevant before. But it sort of is relevant now. And we weren't going to check it against anything. But it turns out to be pretty much good enough.

It has some unfortunate consequences. Like if this is-- well, there's some complicated things you might abuse here. But it doesn't seem to be that bad. It seems certainly better than the server filling up its memory and swapping serving requests all together.

And then we don't include in this computation. Because the only thing we care about here is offloaded the storage of this table and making sure that the only connections that eventually you do get established are legitimate clients. Because therefore, we can say, well, if this client is establishing a million connections to me, I'll stop accepting connections from him.

That's easy enough, finally. The problem is that all these source addresses, if they're spoofed, are hard to distinguish from legitimate clients. Make sense? Yeah.

AUDIENCE: Would you need to store the timestamp?

PROFESSOR: Ahh, so the clever thing, the reason this timestamp is sort of on the slide here, is that when we receive this SNS value in step three, we need to figure out, how do you compute the input to this function F to check whether it's correct? So actually, we take the timestamp from the end of the packet, and we use that inside of this computation.

Everything else we can reconstruct. We know who just sent us the third step and packet. And we have all these fields. And we have our key, which is, again, still secret. And this timestamp just comes from the end of the sequence, from the last 8 bits. And then it might be that we'll reject timestamps that are too old, just disallow old connections. Yeah.

AUDIENCE: So I'm guessing the reason you only use this when you're under attack is because you lose 8 bits of security, or whatever?

PROFESSOR: Yes, it's not great. It has many bad properties. One is you sort of lose 8 bits of security in some sense. Because now the unguessable part is just 24 bits instead of 32 bits. Another problem is what happens if you lose certain packets? So if this packet is lost-- so it's typically, in TCP, there's someone responsible for retransmitting something if a particular packet is lost. And in TCP, if the third packet is lost, then the client might not be waiting for anything. Or sorry, maybe the protocol we're running on top of this TCP connection is one where the server is supposed to say something initially.

So I connect. I just listen. And in the SMTP, for example, the server is supposed to send me some sort of an initial greeting in the protocol. So OK, suppose I'm connecting to an SMTP server. I send my third packet. I think I'm done. I'm just waiting for the server to tell me, greetings as an SMTP server. Please send mail.

This packet could get lost. And in real TCP, the way this gets handled is that the server from step two remembers that, hey, I sent this response. I never heard back, this third thing. So it's the server that's supposed to resend this packet to trigger the client to resend this third packet.

Of course, if the server isn't storing any state, it has no idea what to resend. So this actually makes connection establishment potentially programmatic where you could enter this weird state where both sides are waiting for each other. Well, the server doesn't even know that it's waiting for anything. And the client is waiting for the server. And the server basically dropped responsibility by not storing state. So this is another reason why you don't run this in production mode all the time. Yeah.

AUDIENCE: Presumably also you could have data commissions if you establish two very short-lived connections right after each other from the same host.

PROFESSOR: Absolutely, yeah, yeah. So another thing is, of course, because we gave up on using this ISN old style part, we now give up protection against these multiple connections in a short time period being independent from one another. So I think there's a number of trade-offs. We just talked about three. There's several more things you worry about.

But it's not great. If we could design a protocol from scratch to be better, we could just have a separate nice 64-bit header for this and a 64-bit value for this. And then we could enable this all the time without giving up the other stuff and all these nice things. Yeah.

AUDIENCE: I just had one quick question on the SNS. In step two, [INAUDIBLE], do they have to be the same?

PROFESSOR: This SNS and this SNS?

AUDIENCE: Mhm.

PROFESSOR: Yeah, because otherwise, the server has no way to conclude that this client got our packet. If the server didn't check that this SNS was the same value as before, then these actually would be even worse.

Because I could fake a connection from some arbitrary IP address, then get this response. Maybe I don't even get it, because it goes to a different IP. Then I establish a connection from some other IP address. And then the server is maintaining a whole live connection. Probably a server crosses another side waiting for me to send data and so on.

AUDIENCE: But the timestamp is going to be different, right? So how can the server recalculate that with a new timestamp and null the one before if it doesn't store any state?

PROFESSOR: So the way this works is these timestamps, as I was saying, are course grained. So they're on a scale of minutes. So if you connect within the same minute, then you're in good shape. And if you connect on the minute boundary, well, too bad.

Yet another problem with the scheme-- it's imperfect in many ways. But most operating systems, including Linux, actually have ways of detecting if there's too many entries building up in this table that aren't being completed. It switches to this other scheme instead to make sure it doesn't overflow this table. Yeah.

AUDIENCE: So if the attacker has control of a lot of IP addresses, and they do this, and even if you switch it the same--

PROFESSOR: Yeah, so then actually there's not much you can do. The reason that we were so worried about this scheme in the first place is because we wanted to filter out or somehow distinguish between the attacker and the good guys. And if the attacker has more IP addresses and just controls more machines than the good guys, then he can just connect to our server and request lots of web pages or maintain connections.

And it's very hard then for the server to distinguish whether these are legitimate clients or just the attacker tying up resources of the server. So you're absolutely right. This only addresses the case where the attacker has a small number of IP addresses and wants to amplify his effect.

But it is a worry. And in fact, today it might be that some attackers control a large number of compromised machines, like just desktop machines of someone that didn't patch their machine correctly. And then they can just mount denial of service attacks from this distributed set of machines all over the world. And that's pretty hard to defend against.

So another actually interesting thing I want to mention is denial of service attacks, but in the particular way that other protocols make them worse. I guess other protocols allow denial of service attacks in the first place. I'm sorry. But there are some that are protocols that are particularly susceptible to abuse. And probably a good example of that is, again, this DNS protocol that we were looking at before.

So the DNS protocol-- we still have it here-- involves the client sending a request to the server and the server sending a response back to the client. And in many cases, the response is larger than the request. The request could be just, tell me about mit.edu. And the response might be all the records the server has about mit.edu-- the email address, the mail server for mit.edu, the assigned record if it's using DNS SEC, and so on.

So the query might be 100 bytes. The response could well be over 1,000 bytes. So suppose that you want to flood some guy with lots of packets or lots of bandwidth. Well, you might only be able to send a small amount of bandwidth.

But what you could do is you could fake queries to DNS servers on behalf of that guy. So you only have to send 100 bytes to some DNS server pretending to be a query from that poor guy. And the DNS server is going to send 1,000 bytes to him on your behalf.

So this is a problematic feature of this protocol. Because it allows you to amplify bandwidth attacks. And partly for the same reason we were talking about with TCP's SYN flooding attacks, it's very hard for the server, for the DNS server, in this case, to know whether this request is valid or not. Because there's no authentication or no sort of sequence number exchanges going on to tell that this is the right guy connecting to you, et cetera.

So in fact this is still a problem in DNS today. And it gets used quite frequently to attack people with bandwidth attacks. So if you have a certain amount of bandwidth, you'll be that much more effective if you reflect your attack off of a DNS server.

And these DNS servers are very well provisioned. And they basically have to respond to every query out there. Because if they stop responding to requests, then probably some legitimate requests are going to get dropped. So this is a big problem in practice. Yeah.

AUDIENCE: So if you can still see it on the DNS server, [INAUDIBLE] requests and never reply to--

PROFESSOR: Right, yeah, so it's possible to maybe modify the DNS server to keep some sort of state like this.

AUDIENCE: That's the reason why this still works now, because they don't store state?

PROFESSOR: Yeah, well I think some people are starting to modify DNS server to try to store state. A lot of times, there's so many DNS servers out there that it doesn't matter. Even if you appear to do 10 queries against every DNS server, that's still every packet gets amplified by some significant factor. And they have to respond. Because maybe that client really is trying to issue this query. So this is a problem. Yeah, so you're right, if this was one DNS server, then this would be maybe not as big of a deal.

The problem is also that the root servers for DNS, for example, aren't a single machine. It's actually racks and racks of servers. Because they're so heavily used. And trying to maintain a state across all these machines is probably nontrivial. So as it gets abused more, probably it will be more worthwhile to maintain this state.

I guess a general principle you want to follow in any protocol-- well, might be a good principle-- is to make the client do at least as much work as the server is doing. So here, the problem is the client isn't doing as much work as the server. That's why the server can help the client amplify this effect.

If you were redesigning DNS from scratch, and this was really your big concern, then it'd probably be fairly straightforward to fix this. The client has to send a request that has extra padding bytes just there just wasting bandwidth. And then the server is going to respond back with a response that's at most as big as that.

And if you want a response that's bigger, maybe the server will say, sorry, your padding wasn't big enough. Send me more padding. And this way, you guarantee that the DNS server cannot be used ever to amplify these kinds of bandwidth attacks.

Actually, these kinds of problems happen also at higher levels as well. So in web applications, you often have web services that do lots and lots of computation on behalf of a single request. And there's often denial of service attacks at that level where adversaries know that a certain operation is very expensive, and they'll just ask for that operation to be done over and over again. And unless you carefully design your protocol and application to allow the client to prove that, oh, I'm burning at least as much work as you, or something like this, then it's hard to defend against these things as well. Make sense?

All right, so I guess the last thing I want to briefly touch on about the paper we talked about as well is these routing attacks. And the reason these attacks are interesting is they're maybe popping up a level above these protocol transport level issues. And look at what goes wrong in an application.

And the routing protocol is a particularly interesting example. Because it's often the place where trust and sort of initial configuration gets bootstrapped in the first place. And it's easy to sort of get that wrong. And even today, there's not great authentication mechanisms for that.

Perhaps the clearest example is the DHCP protocol that all of you guys use when you open a computer or connect to some wireless or wired network. The computer just sends out a packet saying, I want an IP address and other stuff. And some DHCP server at MIT typically receives that packet and sends you back, here's an IP address that you should use. And also here's a DNS server you should use, and other interesting configuration data.

And the problem is that the DHCP request packet is just broadcasting on the local network trying to reach the DHCP server. Because you actually don't know what the DHCP is going to be ahead of time. You're just plugging into the network, the first time you've been here, let's say. And your client doesn't know what else to do or who to trust.

And consequently, any machine on the local network could intercept these DHCP requests and respond back with any IP address that the client could use, and also maybe tell the client, hey you should use my DNS server instead of the real one. And then you could intercept those future DNS requests from the client and so on. That make sense?

So I think these protocols are fairly tricky to get right. And on a global scale, the protocols like BGP allow any participant to announce a particular IP address prefix for the world to sort of know about and route packets toward the attacker. There's certainly been attacks where some router participating in BGP says, oh, I'm a very quick way to reach this particular IP address range. And then all the routers in the world say, OK, sure, we'll send those packets to you.

And probably the most frequent abuse of this is by spammers who want to send spam, but their old IP addresses are blacklisted everywhere, because they are sending spam. So they just pick some random IP address. They announce that, oh yeah, this IP address is now here. And then they sort of announce this IP address, send spam from it, and then disconnect. And it gets abused a fair amount this way.

It's sort of getting less now. But it's kind of hard to fix. Because in order to fix it, you have to know whether someone really owns that IP address or not. And it's hard to do without establishing some global database of, maybe, cryptographic keys for every ISP in the world. And it takes quite a bit of effort by someone to build this database.

The same actually applies to DNS SEC as well. In order to know which signature to look for in DNS, you have to have a cryptographic key associated with every entity in the world. And it's not there now. Maybe it'll get built up slowly. But it's certainly one big problem for adopting DNS SEC.

All right, so I guess the thing to take away from this is maybe just a bunch of lessons about what not to do in general in protocols. But also actually one thing I want to mention is that while probably secrecy and integrity are good properties and driving force of higher levels of abstraction, like in cryptographic protocols in the application-- and we'll look at that in next lectures-- one thing that you really do want from the network is some sort of availability and DOS resistance. Because these properties are much harder to achieve at higher levels in the stack.

So you really want to avoid things like maybe these amplification attacks, maybe these SYN flooding attacks, maybe these RST attacks where you can shoot down an arbitrary person's connection. These are things that are really damaging at the low level and that are hard to fix higher up. But the integrity and confidentiality you can more or less solve with encryption. And we'll talk about how we do that in the next lecture on Cerberus. See you guys then.