PROFESSOR: In this class, this semester, the other co-lecturer is going to be James Mickens, who is a visiting professor from Microsoft Research. He'll lecture on some other topics like web security later on. But we'll decide later what's going on exactly, in terms of the lecture split up. We also have four TAs this year, Stephen, Webb, [INAUDIBLE], and James. And hopefully you'll meet them in office hours over the year if you need help.

So the plan for this class is to understand how to build secure systems, why computer systems sometimes are insecure and how we can make them better, and what goes wrong. And in order to do this, there's not really a great textbook about this topic. So instead, what we're going to do is, each lecture other than this one is going to be focused around some research, typically a paper, that we'll assign on the website and you guys should read ahead of time. And there are some question that you should answer in the submission system about the paper. And submit your own question by 10:00 PM before the lecture day.

And then when you come to lecture, we'll actually discuss the paper, figure out, what is the system? What problem does it solve? When does it work? When does it not work? Are these ideas any good in other cases? Et cetera. And hopefully, through these case studies, we'll get some appreciation of how do we actually build systems that are secure.

And we have some preliminary schedule up on the website. If there's other topics you guys are particularly interested in, or if there's particular papers you're excited about, just send us email and we'll see if we can fit them in or do something. We're pretty flexible. So if there's anything that you'd like to hear more about, just let us know.

And in a similar vein, if you ever have a question or if there's some mistake, just interrupt and ask us what's going on in lecture, anytime. Security is, in many ways, all about the details and getting everything right. And I will inevitably make mistakes. So if something doesn't seem right, there's a good chance it's not. And you should just interrupt and ask. And we'll figure out what's going on and what's the right way to do things.

And I guess in terms of the class organization, the other large part of the class, in addition to lectures, is going to be a series of lab assignments. The first one is already posted on the website. And these lab assignments will help you go through understanding the different range of security problems and how do you prevent them in a simple web server. So in lab one, which is out right now, you'll actually take a web server that we give you and find ways to exploit buffer overflow vulnerabilities in it and take control of this website by just sending it carefully-crafted requests and packets. And in other labs, you'll look at ways to defend the web server, to find bugs in the code, to write worms that run in the user's browser, and other kinds of interesting security problems.

One thing that surprises many students is that every lab uses a different language. So lab one is all about C and Assembly. Lab two involves a lot of Python coding. Lab three will be something else. Lab five will be JavaScript. And so on. This is sort of inevitable. And I sort of apologize ahead of time that you're going to have to learn all these languages if you haven't seen them already.

In some ways it's useful, because the real world is like this. All the systems are complicated and composed of different parts. And in the long run, it'll be useful for you, for your moral character or something like that, to learn this stuff. But it will take some preparation, especially if you haven't seen these languages before. It might be helpful to start early.

In particular, lab one is going to rely on a lot of subtle details of C and Assembly code that we don't really teach in other classes here in as much detail. So it's probably a good idea to start early. And we'll try to get the TAs to hold office hours next week where we'll do some sort of a tutorial session where we can help you get started with understanding what a binary program looks like, how to disassemble it, how to figure out what's on the stack, and so on. All right.

And I guess the one other thing, we're actually videotaping lectures this year. So you might be able to watch these online. We'll post them as soon as we get them ourselves from the video people.

And the last bit of administrivia is you should, if you have questions online, we're using Piazza, so I'm sure you've used this in other classes.

All right. So before we dive into security, I need to tell you one thing. There is a sort of rules that MIT has for accessing MIT's network when you're, especially, doing security research or playing with security problems, you should be aware that not everything you can technically do is legal. And there's many things that you will learn in this class that are technically possible. We'll understand how systems can be broken or compromised. Doesn't mean you should go out and do this everywhere. And there's this link in the lecture notes we'll post that has some rules that are good guidelines. But in general, if you're in doubt, ask one of the lecturers or a TA as to what you should do. And hopefully it's not too puzzling, what's going on.

All right. So any questions about all this administrivia before we dive in? Feel free to ask questions. OK.

So what is security? So we'll start with some basic stuff today. And we'll look at just some general examples of why security is hard and what it means to try to build a secure system. Because there's not really a paper, this will not have sort of deep intellectual content, maybe, but it'll give you some background and context for how to think about secure systems.

So security, in general, is all about achieving some goal when there is an adversary present. So think of it as there's some bad guy out there that wants to make sure you don't succeed. They want to steal your files. They want to delete your entire hard drive contents. They want to make sure nothing works and your phone doesn't connect, all these things, right? And a secure system is one that can actually do something, regardless of what the bad guy is trying to do to you.

So it's kind of cool that we can actually potentially build systems that are resilient to a whole range of bad guys, adversaries, attackers, whatever you want to call them. And we can still build computer systems that allow us to get our work done.

And the general way to think about security is sort of break it up into three parts. One part is roughly the policy that you want your system to enforce. This is roughly the goal that you want to achieve. Like well, maybe, only I should be able to read the grades file for 6.858. Or maybe the TAs as well, and all the co-lecturers, et cetera. But there is some statement about what I want my system to be able to do.

And then, if you want sort of think about what kinds of policies you might write, typical ones have to do with either confidentiality of data, so the grades file is only accessible to the 6.858 course staff. Another example of a security policy has something to do with integrity. For example, only the course staff can also modify the grades file. Or only the course staff can upload the final grades to the registrar's office. That'll be great.

Then you can also think about things like availability. So for example, a website should be available, even if the bad guys try to take it down and mount some sort of a DOS-- Denial of Service-- attack on it.

So this is all well and good. So these are the policies that we might actually care about from a system. But because it's security, there's a bad guy involved. We need to understand, what are we thinking the bad guy is going to do? And this is typically what we call a threat model. And this is basically just a set of assumptions about the bad guy or adversary.

And it's important to have some sort of assumptions about the bad guy because, if the bad guy is omnipresent and is everywhere at once and you can do anything they want, it's going to be hard to achieve some semblance of security. So for example, you probably want to assume the bad guy doesn't exactly know your password, or they don't actually have physical access to your phone and your keys and your laptop. Otherwise, it's going to be hard to make some sort of progress in this game. And turns out that while this is actually quite tricky to come up with, but I guess one general rule is it's much better err on the side of caution and being conservative in picking your threat model, because bad guy might always surprise you in terms of what they might be able to do in practice.

And finally, in order to achieve security, in order to achieve our goal under the set of assumptions, we're going to look at some mechanism. And this is the, basically, software or hardware or whatever part of system design, implementation, et cetera, that's going to try to make sure our policy is followed as long as the bad guy follows the threat model. So the end result is that, as long as our threat model was correct, hopefully we'll satisfy our policy. And it has to be the case that the mechanism doesn't screw up. Make sense? Fairly high level story about how to think about this kind of stuff.

So why is this so hard, right? It seems like a simple plan. You write down these three things, and you're off and running. But in practice, as you, I'm sure, have seen in the world, computer systems are almost always compromised in some way or another. And break ins are pretty commonplace. And the big reason why security tends to be a difficult problem is because what we have here is sort of, this will be familiar to those of you took 6.033, this is a negative goal, meaning that we have to make sure our security policy is followed regardless of what the attacker can do.

So just by contrast, if you want to build a file system, and you want to make sure that my TAs can access the grades file, that's pretty easy. I just ask them, hey, can you guys test and see? Can you access the grades file? And if they all can access it, done. The system works. But if I want to say that no one other than the TAs can access the grades file, this is a much harder problem to solve, because now I have to figure out what could all these non TA people in the world to try to get my grades file, right? They could try to just open it and read it. Maybe my file system will disallow it. But they might try all kinds of other attacks, like guessing the password for the TAs or stealing the TAs laptops or breaking into the room or who knows, right?

This is all stuff that we have to really put into our threat model. Probably for this class, I'm not that concerned about the grades file to worry about these guys' laptops being stolen from their dorm room. Although maybe I should be. I don't know. It's hard to tell, right? And as a result, this security game is often not so clear cut as to what the right set of assumptions to make is. And it's only after the fact that you often realize, well should have thought of that.

All right. And sort of, as a result, this is very much an iterative process. And the thing you end up realizing at every iteration is, well, here's the weakest link into my system. Maybe I got the threat model wrong. Maybe my mechanism had some bugs in it because it's a software and it's going to be large systems. They'll have lots of bugs. And you sort of fix them up. You change your threat model a bit. And you iterate and try to design a new system, and hopefully, make things better.

So one possible interpretation of this class-- well, one danger-- is that you come away thinking, man, everything is just broken. Nothing works. We should just give up and stop using computers. And this is one possible interpretation. But it's probably not quite the right one. The reason this is going to come up or you're going to think this way is because, throughout this class, we're going to look at all these different systems, and we're going to sort of push them to the edge.

We're going to see, OK, well, what if we do this? Is it going to break? What if we do that? Is it going to break then? And inevitably, every system is going to have some sort of a breaking point. And we'll figure out, oh hey. This system, we can break it in if we push this way. And this system doesn't work under these set of assumptions. And it's inevitable that every system will have a breaking point. But that doesn't mean that every system is worthless. It just means you have to know when to use every system design. And it's sort of useful to do this pushing exercise to find the weaknesses so that you know when certain ideas work, when certain ideas are not applicable.

And in reality, this is a little more fuzzy boundary, right? The more secure you make your system, the less likely you'll have some embarrassing story on the front page of New York Times saying, your start up company leaked a million people's social security numbers. And then you pay less money to recover from that disaster.

And I guess one sort of actually positive note on security is that, in many ways, security enables cool things that you couldn't do before, because security, especially mechanisms, that allow us to protect against certain classes of attacks, are pretty powerful. As one example, the browser used to be fairly boring in terms of what you could do with it. You could just view web pages, maybe run some JavaScript code in it. But now there's all these cool mechanisms we'll learn about in a couple of weeks that allow you to run arbitrary x86 native code in the web browser and make sure it doesn't do anything funny to your machine. And it can send-- and there's a technique or system called Native Client from Google that actually allows us to do this securely.

And before, in order to run some native game on your machine, you'd have download and install it, click on lot's of dialogue boxes, say yes, I allow this. But now, you can just run it in a browser, no clicking required. It just runs. And the reason it's so easy and powerful is that our security mechanism can sandbox this program and not have to assume anything about the user choosing the right game and not some malicious game to play in their computer, or some other program to run. So in many ways, good security mechanisms are going to enable constructing cool new systems that weren't possible to construct before.

All right. Make sense? Any questions about this story? All right.

So I guess in the rest of the lecture, I want to go through a bunch of different examples of how security goes wrong. So, so far, we've seen how you can think of it. But inevitably, it's useful to see examples of what not to do so that you can have a better mindset when you're approaching security problems. And in this sort of breakdown of a security system, pretty much every one of these three things goes wrong. In practice, people get the policy wrong, people get the threat model wrong, and people get the mechanism wrong.

And let's, I guess, start with policies and examples of how you can screw up a system's policy. Maybe the cleanest or sort of simplest example of this are account recovery questions. So typically, when you sign into a website, you provide a password. But what happens if you lose your password? Some sites will send you email if you lose your password with a link to reset your password. So it's easy enough, if you have another email address. But what if this is your email provider?

So at least, several years ago, Yahoo hosted email, webmail, for anyone on the internet. And when you forgot your Yahoo password, they couldn't really send you email because you couldn't get it. So instead, they had you register a couple of questions with them that hopefully only you know. And if you forget your password, you can click on a link and say, well, here's the answers to my questions. Let me have my password again.

And what turns out to be the case is-- well, some people failed to realize is that this changes your policy, because before, the policy of the system is people that can log in are the people that know the password. And when you introduce these recovery questions, the policy becomes, well, you can log in if you know either the password or those security questions. So it strictly weakens the security of your system. And many people have actually taken advantage of this.

One sort of well known example is, I think a couple years ago, Sarah Palin had an email account at Yahoo. And her recovery questions were things like, well, where'd you go to school? What was your friend's name? What's your birthday? Et cetera. These were all things written on her Wikipedia page. And as a result, someone can quite easily, and someone did, actually, get into her Yahoo email account just by looking up on Wikipedia what her high school was and what her birthday was. So you really have to think carefully about the implications of different security policies you're making here.

Perhaps a more intricate and, maybe, interesting example, is what happens when you have multiple systems that start interacting with one another. So there's this nice story about a guy called Mat Honan. Maybe you read this story a year or two ago. He's a editor at this wired.com magazine. And had a bit of a problem. Someone basically got into his Gmail account and did lots of bad things.

But how did they do it, right? So it's kind of interesting. So all parties in this story seem to be doing reasonable things. But we'll see how they add up to something unfortunate.

So we have Gmail. And Gmail lets you reset your password if you forget, as do pretty much every other system. And the way you do a reset at Gmail is you send them a reset request. And what they say is, well, you weren't going to do this recovery questions, at least not for this guy. What they do is they send you a recovery link to a backup email address, or some other email address that you have. And helpful, they actually print the email address for you. So for this guy's account, someone went and asked Gmail to reset the password. And they said, well, yeah. Sure. We just sent the recovery link to this email, foo@me.com, which was some Apple email service.

OK, but the bad guy doesn't have access to me.com, either. But they want to get this password reset link to get access to Gmail. Well, the way things worked was that, in Apple's case, this me.com site, allowed you to actually reset your password if you know your billing address and the last four digits of your credit card number. So it's still not clear how you're going to get this guy's-- well, home address, maybe you could look it up somewhere. This guy was a well known person at the time. But where do you get the last four digits of his credit card number? Well, not clear, but let's keep going further.

So you need to send these things to me.com to get access to his email account there. Well, it turns out this guy had an account at Amazon, which is another party in this story. Amazon really wants you to buy things. And as a result, they actually have a fairly elaborate account management system. And in particular, because they really want you to buy stuff, they don't require you to sign in in order to purchase some item with a credit card.

So I can actually go on Amazon, or at least at the time, I was able to go on Amazon and say, well, I'm this user. And I want to buy this pack of toothbrushes. And if I wanted to use the saved credit card number in the guy's account, I shouldn't be able to do this. But if I just was providing a new credit card, what Amazon would do is, they can actually add a new credit card to some guy's account. So that seems not too bad, right? I'm basically ordering toothbrushes through one of your Amazon accounts. But it's not your credit card anyway. It's just my credit card number being used. So it's not clear how things go wrong yet.

But Amazon had another interface. All these are complicated systems. And Amazon had an interface for password reset. And in order to reset a password in Amazon, what you had to provide is just one of the user's credit card numbers. So I can order stuff and add a credit card number to your account. And then I can say, hey, I want to reset my password. This is one of my credit card numbers. And this, in fact, worked. So this is where the bad guy got a hold of this guy's, Mat's, Amazon account.

But OK. How do you fish out the credit card number for resetting Apple's site? Well, Amazon was actually very careful. Even if you break into someone's Amazon account, it will not print you the saved credit card numbers from that person. But it will show the last four digits. Just so you know which credit card you're talking about. So you can list all the credit cards, other than the one you added. You can then go and break into me.com. You can click on this link and get access to the guy's Gmail account.

This is all very subtle stuff. And in isolation, each system seems to be doing somewhat sensible things. But it's actually quite hard to reason about these vulnerabilities and weaknesses unless you have this whole picture explained to you and you've sort of put all the pieces together. So this is actually fairly tricky stuff. And unfortunately, well, much like for every one of these three categories, the answer for how to avoid this is often think hard and be careful.

I guess the one general plan is, be conservative in terms of what you set your policy to be, to maybe not depend on things other sites might reveal. So well, I'm not sure if any really great advice would have prevented this problem. But now you know. And now you'll make other mistakes. There's many other examples of policies going wrong and allowing a system to be compromised.

That's interesting enough. But let's look at how people might screw up threat models. So let me turn off this blue square. OK. So what are examples of threat models that go wrong?

Well, probably a big one in practice is human factors. So we often make assumptions about what people will do in a system, like they will pick a good, strong password, or they will not click on random websites that they get through email and enter their password there. So these are-- well, as you probably suspect, and in practice, happens to be the case, these are not good assumptions in all cases. And people pick bad passwords. And people will click on random links. And people will enter their password on sites that are actually not the right site at all. And they will not be paying a lot of attention.

So you probably don't want to have threat models that make very strong assumptions about what humans will do because inevitably, something will go wrong. Make sense? Any questions? All right.

Another sort of good thing to watch out in threat models is that they sometimes change over time. Or whether something is a good assumption or not changes over time. One example of this is actually at MIT in the mid '90s-- mid '80s, actually-- Project Athena developed this system called Kerberos. And we'll read about this in a couple of weeks in this class. And at the time, they were sort of figuring out, well, Kerberos is going to be based on cryptography. So we need to pick some size keys to make sure they're not going to be guessed by arbitrary people. And they said, OK. Well you know, 56-bit keys, at the time, for this cypher called DES, seemed like a plausible size. Maybe not great, but certainly not entirely unreasonable. And this was in the mid '80s.

But then you know, this system got popular and got used a lot. MIT still uses it. And they never really went back to seriously revisit this assumption. And then, a couple years ago, a group of 6.858 students figured out that actually, yeah, you can just break this, right? It's easy enough to enumerate all the 256 keys these days. Computers are so fast, you can just do it. And as a result, they were able to, with the help of some hardware from a particular web service-- we'll have some links the lecture notes-- they were able to get, basically, anyone's Kerberos account key in roughly a day.

And so this assumption was good in the mid 1980s. No longer a good assumption today. So you really have to make sure your assumptions sort of keep up with the times.

Maybe a more timely example is, if your adversary-- or if you're worried about government attacks, you might realize that you shouldn't trust hardware even these days, right? There was all these revelations about what the NSA is capable of doing. And they have hardware back doors that they can insert into computers. And maybe up until a couple years ago, well, who knows? I guess we didn't know about this stuff. So maybe it was a reasonable assumption to assume your laptop is not going to be compromised physically, the hardware itself.

But now you know. Actually, if you're worried about the government being after you, you probably have a much harder problem to deal with because your laptop might be compromised physically, regardless of what you install in it. So we really have to be careful with your threat model and really sort of balance it against who you think is out to get you. I think it's going to be a very expensive proposition if you're going to try to protect yourself from the NSA, really. On the other hand, if you're just protecting yourself from random other students that are, I don't know, snooping around in your Athena home directory or whatnot, maybe you don't have to worry about this stuff as much. So it's really a balancing game and picking the right threat model.

Another example of a bad threat model shows up in the way secure websites these days check certificates of a website that you're connecting to. So in this SSL protocol or TLS, when you connect to a website and it says HTTPS-- we'll talk much more about this in later lectures-- but what happens is that the site you're connecting to presents you a certificate signed by one of the certificate authorities out there that attests that, yep, this key belongs to Amazon.com.

And architecturally, the sort of mistake or the bad threat model that these guys assumed is that all these CAs are going to be trustworthy. They will never make a mistake. And in fact, the way system works is that there's hundreds of these CAs out there. The Indian postal authority, I think, has a CA. The Chinese government has a CA. Lots of entities are certificate authorities in this design. And any of them can make a certificate for any host name or a domain name.

And as a result, what happens if you're a bad guy, if you want to compromise Gmail or if you want to impersonate Gmail's website, you just have to compromise one of these certificate authorities. And it turns out the weakest link is probably some poorly run authority somewhere in some, you know, not particularly up to date country. Who knows, right? And as a result, it's probably a bad assumption to build a system-- or it's a bad idea to build a system around the assumption that you'll manage to keep all 300 certificate authorities spread out around the globe perfectly secure. But yet, that's the assumption underpinning the security mechanism of today's SSL protocol used by web browsers.

And there's sort of many other, I guess, examples that are things you might not have thought of. Another sort of amusing example from the 1980s was DARPA. This defense agency, at the time, really wanted to build secure operating systems. And they actually went so far as to get a bunch of universities and researchers to build secure OS prototypes. And then they actually got a red team, like a team of bad guys pretending to be the attackers, and told them, well, go break into these secure operating systems any way you can. We actually want to know, is it secure?

And it's kind of amusing, some of the surprising ways they compromised the systems. One was that there was this OS research team that seemed to have a perfectly secure OS, but it got compromised. And the way it happened is that the server in which the source code of the operating system was stored was some development machine in someone's office that wasn't secured at all. But that had all the source code. So the bad guys broke into that server. It was not protected very well. Changed the source code of the operating system to introduce a back door. And then, when the researchers built their operating systems, well, it had this back door. And the bad guys were able to break in.

So you really have to think about all the possible sort of assumptions you're making about where your software is coming from, about how the bad guy can get in, in order to make sure your system is really secure. And there's many other examples in lecture notes, if you want. So I'm using anecdotes. You can page through those.

Probably the most pervasive problem that shows up, of course, is in mechanisms, though. And in part, it's because mechanisms are the most complicated part of the story. It's the entirety of all the software and hardware and all that sort of system components that make up what is trying to enforce your security policy. And there's no end of ways in which mechanisms can fail.

And, partly as a result, much of this class will focus pretty heavily on mechanisms and how do you make mechanisms that are secure, that provide correct enforcement of security policies. And we'll talk about threat models and policies as well. But turns out it's much easier to make clean, sort of crisp statements about mechanisms and ways they work and don't work, as opposed to policies and threat models which, really, you have to figure out how to fit them into a particular context where you're using a system.

So let's look at some examples of, I guess, mechanism bugs. One that you might have heard in the last couple of days was a problem in the security mechanism in Apple's cloud infrastructure called iCloud. Well actually, any one of you that has an iPhone might be using this iCloud service. They basically provide storage for files and let you find your iPhone if you lose it, and probably lots of other useful features. And I think it's some relative of this me.com service that was implicated in this scheme a couple years back.

And the problem that someone discovered in this iCloud service is that they didn't enforce the same sort of mechanism at all interfaces. OK, so what does iCloud look like? Well, it basically provides lots of services for the same sort of set of accounts. So maybe you have your file storage on iCloud. Maybe you have your photo sharing. Maybe you have other interfaces. And one of the interfaces into iCloud-- these are all sort of at different APIs that they provide-- was this feature to find my iPhone, I think.

And all these interfaces want to make sure that you are the right user, you're authenticated correctly. And unfortunately, the developers all this iCloud system, you know it's a giant piece of software. I'm sure lots of developers worked on this. But on this particular interface, the find my iPhone interface, when you tried to log in with a username and password, they didn't keep track of how many times you tried to log in. And the reason is important is that, as I mentioned earlier, humans are not that great at picking good passwords.

So actually building a system that authenticates users with passwords is pretty tricky. We'll actually read a whole paper about this later on. But one good strategy is, there's probably a million passwords out there that will account for 50% percent of accounts. So if you can guess, make a million attempts at someone's account, then there's a good chance you'll get their password because people actually pick predictable passwords. And one way to try to defeat this is to make sure that your system doesn't allow an arbitrary number of attempts to log in to an account. Maybe after three or 10 tries, you should say, well, you've had enough tries. Time out. You can try again in 10 minutes or in an hour.

And this way you really slow down the attacker. So they can only make a handful of guesses a day, instead of millions of guesses. And as a result, even if you have not the greatest of passwords, it's going to be pretty hard for someone to guess it.

What would happen is that iCloud had this password guessing prevention or, basically, back off, on some interfaces, like if you tried to log in through other interfaces and you failed 10 times, it would say, well, sorry. You have to wait until you try again. But on this find my iPhone interface, they forget this check. That's probably, you know, some guy just forgot to call this function on this API. But the result is that, for the same set of accounts, a bad guy would be able to now guess your password through this interface at millions of attempts per day easily, because this is just limited up to how fast they can send packets to this iCloud thing.

And they can probably guess your password with pretty good accuracy, or with pretty good success rate, after making many guesses. And this led to some unfortunate break ins. And people's confidential data got stolen from this iCloud service. So this is sort of an example of you had the right policy. Only the user and the right password would get you access to the files. You even had the right threat model that, well, the bad guy might be able to guess the password. So we'll have to break limit the number of guess attempts. But he just screwed up, like the mechanism had a bug in it. He just forgot to enforce this right policy and mechanism at some interface.

And this shows up again and again in systems, where just made a mistake and it has pretty drastic effects on the security of the overall system. This make sense? Any questions so far? All right. OK.

So another example-- this is sort of an example of you forget to check for password guessing attempts. There's many other things you can forget. You could forget to check for access control altogether. So one example is, Citibank had a website-- actually, still has a website that allows you to look at your credit card account information. So if you have a credit card with Citibank, you go to this website, it tells you, yeah, you have this credit card. Here's all the charges, all this great stuff.

And the workflow a couple of years ago was that you go to some site, you provide a log in username and password, and you get redirected to another URL, which is something like, I don't know, I'm guessing, but basically like citi.com/account?id= you know, whatever, one two three four. And it turns out that some guy figured out, well, if you change this number, you just get someone else's account. And it's not clear quite how to think of this.

One possibility is that these guys were just thinking right, but they, again, forgot to check a function in this account page that, not only do I have a valid ID number, but it's also the ID number of the guy that's currently logged in. It's an important check to me. But it's easy to forget.

Another thing is, maybe these guys were thinking, no, no one could hit URLs. Maybe they had a bad threat model, right? Maybe they're thinking, the URL-- if I don't print this URL, no one can click on it. It's like a bad threat model. So maybe that's-- well, it's hard to tell exactly what went wrong. But anyway, these mistakes do happen. And they show up a lot. So easy to have small, seemingly, bugs in your mechanism lead to pretty unfortunate consequences.

Another example that's not so much in missing checks is a problem that showed up on Android phones a couple of months ago. Maybe I'll use this board over here. So the problem was related to Bitcoin, which is this-- well, I'm sure you've heard-- this electronic currency system that's pretty popular these days. And the way that Bitcoin works, at a very high level, is that your balance of Bitcoins is associated with a private key. And if you have someone's private key you can, of course, spend their Bitcoins.

So the security of Bitcoin relies quite heavily on no one else knowing your private key. It's kind of like a password, except it's even more important, because people can probably make lots of guesses at your private key. And there's no real server that's checking your key. It's just cryptography. So any machine can try to make lots of guesses at your private key. And if they guess it, then they can transfer your Bitcoins to someone else. And as a result, it's critically important that you generate good, random keys that no one else can guess.

And there are people using Bitcoin on Android. And the Android applications for Bitcoin were getting random values for these keys using this Java API called SecureRandom(), which sounds great, but as people figured out, well, OK. So what it is, right, it doesn't really get real random numbers. Inside of it, there's this construction called Pseudorandom Number Generator, or PRNG that, given a particular seed value, like you get maybe a couple of hundred bits of randomness and you shove it into this PRNG, you can keep asking it for more randomness and sort of stretch these random bits into as many random bits as you want. So you see them initially, and then you can generate as many random bits as you want.

And for various cryptographic reasons I won't go into here, it actually works. If you give it a couple of hundred really good random bits initially, it's going to be very hard for anyone to predict what the pseudorandom values it's generating are. But the problem is that this Java library had a small bug in it. In some set of circumstances, it forgot to initialize the PRNG with a seed, so it was just all zeros, which means that everyone could just figure out what your random numbers were. If they start with zeros, they'll produce the same random numbers as you, which means they'll produce the same private key as you. So they can just generate the same private key and transfer your Bitcoins.

So this is, again, a small or not small bug, depending on, I guess, who is asking. But nonetheless, right? Another example of small programming mistakes leading to pretty catastrophic results. Lot's of people got their Bitcoin balances stolen because of this weakness. Of course, the fix is pretty simple at some level. You change the Java implementation of SecureRandom() to always seed this PRNG with random input bits. And then, hopefully, you're in good shape. But still, that's yet another example of mechanism failure. Yeah?

AUDIENCE: Just to be clear, is this a different attack from the DSA signature randomness?

PROFESSOR: Well yeah. So the actual problem is a little bit more complicated, as you're hinting at. The problem is, even if you didn't generate your key on the Android device in the first place, the particular signature scheme used by Bitcoin assumes that every time you generate a new signature with that key, you use a fresh, what's called a nonce, for generating that signature. And if you ever generate two signatures with the same nonce, then someone can figure out what your key is.

The story is pretty similar. But the details are a little different. So yeah, even if you actually generated your key somewhere else and your key was great, it's just that every time you generate a signature, you would-- and you generated two signatures with exactly the same nonce, or random value, someone could apply some clever math to your signatures and sort of extract your public key out of it. Or private key, more importantly.

All right. Other questions about these problems, examples, et cetera? All right.

So I guess, one thing I wanted to point out is that actually, well, as you're starting to appreciate, is that in computer security, almost every detail has a chance of really mattering. If you screw up almost something seemingly inconsequential, like forgetting to check something, or this, or forgetting to initialize the random seed, it can have pretty dramatic consequences for the overall system. And you really have to be very clear about, what is the specification of your system? What is it doing? Exactly what are all the corner cases?

And a good way to sort of think of breaking a system or, conversely, figure out if your system is secure, is to really push all the edge cases, like what happens if my input is just large enough? Or what is the biggest or the smallest input? What is the sort of strangest set of inputs I could provide to my program and push it in all these corner cases?

One example of this ambiguity, sort of a good example to keep in mind, is how SSL certificates, again, encode names into the certificate itself. So this is a different problem than the problem about the certificate authorities being trusted. So these SSL certificates are just sequences of bytes that a web server sends to you. And inside of this SSL certificate is the name of the server you're connecting to, so something like Amazon.com. You know, you can't just put down those bytes. You have to encode it somehow and specify, well, it's Amazon.com. And that's the end of the string.

So in SSL certificates, they use a particular encoding scheme that writes down Amazon.com by first writing down the number of bytes in the string. So you first write down, OK. Well, I'm going to have a 10 byte string called Amazon.com. That's actually 10 bytes. Great. OK.

So this is like-- in the SSL certificate, somewhere in there, there is this byte 10 followed by 10 bytes saying what the host name is. And there's other stuff afterwards, right, and before. And when a browser takes it, well, the browser is written in C. And the way C represents strings is by null terminating them. So in C, a string doesn't have a length count. Instead, it has all the bytes. And the end of the string is just the byte zero. And in C, you write it with a backslash zero character.

So this is in memory in your browser. Somewhere in memory there's this string of 11 bytes, now, with an extra zero at the end. And when a browser interprets this string, it just keeps going until it sees an end of string marker, which is a zero byte. OK. So, what could go wrong? Any guesses? Yeah?

AUDIENCE: You have a zero in the middle [INAUDIBLE]?

PROFESSOR: Yes. This is great. All right. So, this is actually a bit of a discontinuity in terms of how this guy represents strings and this guy. So suppose that I own the domain foo.com. So I can get certificates for anything dot foo dot com. So what I could do is ask for a certificate for the name amazon.com0x.foo.com. That's a perfectly valid string. It has a bunch of bytes. I guess it's 10, 11 12 13, 14, 15, 16, there's another four, 20, right? So this is 20 byte name with these 20 bytes.

So it used to be that if you go to a certificate authority, in many cases, you could say, hey, I own foo.com. Give me a certificate for this thing. And they'd be perfectly willing to do it because it's a subdomain of foo.com. It's all yours. But then, when a browser takes this string and loads it in memory, well, what it does is the same thing it did here. It copies the string. amazon.com0x.foo.com. It'll dutifully add the terminating zero at the end. But then, when the rest of the browser software goes and tries to interpret the string at this memory location, it'll keep going up until it gets to zero and say, OK well, that's the end of the string. So this is Amazon.com. That's it.

So this sort of disconnect between how C software and how SSL certificates represent names led to some unfortunate security problems. This was actually discovered a number of years ago now by this guy, Moxie Marlinspike. But it's a fairly clever observation. And these kinds of encoding bugs are actually also pretty common in lots of software because, unless you're very diligent about exactly how you encode things, there might be different ways of encoding. And whenever there's disagreement, there's a chance the bad guy can take advantage of this. One system thinks that's a fine name. Another thinks that's not, something else. So these are good places to sort of push a system to see how it might break. That make sense?

All right. So maybe the last example of mechanism failure I'm going to talk about today is a reasonably popular one. It's this problem or buffer overflows. So some of you have seen this in, or at least at some level, in 6.033, if you did the undergrad course. But for those of you that have forgotten or haven't taken oh three three, we'll sort of go over buffer overflows in more detail. And this will be, actually, quite critical for you guys, because lab one is all about buffer overflows. And you're going to be exploiting these vulnerabilities in a somewhat real web server.

So let's figure out, what is the setting? What are we talking about here? So the setting we're going to be considering is a system which has, let's say, a web server. So what we have is, we have some computer out there that has a web server on it. And the web server is a program that is going to accept connections from the outside world, take requests-- which are basically just packets-- and somehow process them, and do some checking, probably. If it's an illegal URL or if they're trying to access a file they are not authorized to access, the web server is going to return an error. But otherwise, it's going to access some files, maybe on disk, and send them back out in some sort of a reply.

So this is a hugely common picture, almost any system you look at. What's the policy? Or what's the threat model? So this is a bit of a problem in many real world systems, namely that it's actually pretty hard to pin down what is the exact policy or threat model that we're talking about. And this sort of imprecision or ambiguity about policies, threat models, et cetera, is what sometimes leads to security problems. Not in this particular case, but we'll see.

But maybe just to give you a sense of how to think of a typical web server in the context of this policy, threat model kind of stuff, is that well, probably the policy is, the web server should do what the programmer intended it to do. It's a little vague. But that's probably what's going on because anything more specific, as well, the web server should do exactly what the code does, is going to be a bit of an [INAUDIBLE] And if your code has a bug, well, your policy says, well, that's exactly what I should do. I should follow the bug.

So it's a little hard to state a policy precisely, but in this case, let's go with some intuitive version of, well, the web server should do what the programmer wanted it to do. And the threat model is probably, the attacker doesn't have access to this machine, can't log in to it remotely, doesn't have physical access to it, but can send any packet they want. So they're not restricted to certain kinds of packets. Anything you can shape and sort of deliver to this web server, that's fair game. Seems like a reasonable threat model, in practice, to have in mind.

And I guess the goal is that this web server shouldn't allow arbitrary stuff to go wrong here. I guess that sort of goes along with what the programmer intended. The programmer probably didn't intend any request to be able to access anything on the server. And yet, it turns out if you make certain kinds of mistakes in writing the web server software, which is basically the mechanism here, right? The web server software is the thing that takes a request and looks at it and makes sure that it's not going to do something bad, sends a response back if everything's OK. The web server in this mechanism. It's enforcing your policy.

And as a result, if the web server software is buggy, then you're in trouble. And one sort of common problem, if you're writing software in C which, you know, many things are still written in C and probably will continue to be written in C for a while, you can mismanage your memory allocations. And as we saw in this SSL certificate naming example, even sort of a single byte can really make a huge difference, in terms of what goes on. And I guess for this example, we'll look at a small piece of code that's not quite a real web server. In the lab, you'll have this whole picture to play with. But for lecture, I just want to give you a simplified example so we can talk about what's sort of at the core of what's going wrong.

And, in particular, if this system wakes up, I will show you sort of a very small C function. And we can sort of see what goes wrong if you provide different inputs to that piece of code. All right.

So the C function that I have in mind is this guy. Somewhere here. Oh, yeah. It's coming on. All right. So here's the sort of program I'm talking about, or I want to use as an example here. So this program is just going to read a request. And you can sort of imagine it's going to read a request from the network. But for the purposes of this example, it's just going to read a request from whatever I'm typing in on the keyboard. And it's going to store it in a buffer here. And then it's going to parse it is an integer and return the integer. And the program will then print whatever integer I get back.

It's like far from a web server. But we'll at least see some basics of how buffer overflows work and what goes wrong. So let's see actually what happens if we run this program. So I can compile it here. And actually, you can sort of see the-- it's already telling me what I'm screwing up, right? The get function is dangerous and should not be used. And we'll see in a second why the compiler is so intent on telling me this. And it actually is true. But for now, suppose we're a happy go lucky developer that is willing to ignore this warning.

So OK. I run this redirect function, I provide some input, and it works. Let's see if I provide large inputs. If I type in some large number, well, at least it gives me some large number. It basically maxes out to two to the 31 and prints that and doesn't go any higher. So that's maybe not disastrous, right? Whatever. You provided this ridiculously large number. You got something didn't quite work.

It's not quite a problem yet. But if we provide some really large input, we might get some other problem, right? So suppose I provide in a lot of by 12 I just provided things that are not numbers. It prints zero. That's not so bad. But suppose I'm going to paste in a huge number of As. OK, so now the program crashes. Maybe not too surprising. So if it was the case that if I send a bad request to the web server, it just doesn't get back to me or doesn't send a reply, that would be fine. But we'll sort of look inside and see what happens, and try to figure out how we can actually take advantage of this crash to maybe do something much more interesting, or, well, much more along with what a hacker might be interested in doing.

So to do this, we're going to run this program under a debugger. You'll get super familiar with this in lab one. But for now, what we're going to do is set a breakpoint in that redirect function. And we're going to sort of run along and see what happens. So when I run the program, it's going to start executing in the main function. And pretty quickly, it calls redirect. And the debugger is now stopped at the beginning of redirect. And we can actually see what's going on here by, for example, we can ask it to print the current CPU registers.

We're going to look at really low level stuff here, as opposed to at the level of C source code. We're going to look at the actual instructions that my machine is executing because that's what really is going on. The C is actually maybe hiding some things from us. So you can actually print all the registers. So on x86, as you might remember. Well, on [INAUDIBLE] architecture, there's a stack pointer. So let me start maybe drawing this diagram on the board so we can try to reconstruct what's happening.

So what's going on is that my program, not surprisingly, has a stack. On x86, the stack grows down. So it sort of is this stack like this. And we can keep pushing stuff onto it.

So right now, the stack pointer points at this particular memory location FFD010. So some value. So you can try to figure out, how did it get there? One way to do it is to disassemble the code of this redirect function. Is this going to work better? Really? Convenience variable must have integer value. Man. What is going on with my debugger?

All right. Well, we can disassemble the function by name. So this is what the function is doing. So first off, it starts by manipulating something with this EBP register. That's not super interesting. But the first thing it does after that is subtract a certain value from the stack pointer. This is, basically, it's making space for all those variables, like the buffer and the integer, i, we saw in the C source code.

So we're actually, now, four instructions into the function, here. So that stack pointer value that we saw before is actually already in the middle, so to say, of the stack. And currently, there's stuff above it that is going to be the buffer, that integer value, and actually, also the return address into the main function goes on the stack, as well. So somewhere here, we'll have the return address. And we actually try to figure out, where are things on the stack?

So we can print the address of that buffer variable. So the buffer variable is at address D02C. We can also print the value of that integer, i. That guy is at D0AC. So the i is way up on the stack. But the buffer is a bit lower.

So what's going on is that we have our buffer here on the stack, and then followed above by i and maybe some other stuff, and then finally, the return address into the main function that called redirect. And the buffer is-- this is going, the stack is growing down. So these are higher addresses. So what this means is that the buffer-- we actually have to decide, where is the zeroth element of the buffer, and where is the 128th element of this buffer? So where does the zeroth element of the buffer go? Yeah?

Should be at the bottom, right, because yeah, higher elements just keep going up. So buff of zero is down here. It just keeps going on. And buff of 127 is going to be up there. And then we'll have i and other stuff. OK.

Well, let's see what happens now if we provide that input that seemed to be crashing it before. So I guess one thing we can actually do before this is to see whether we can actually find this return address. Where it actually happens to live is at the EBP pointer. This is just a convenient thing in the x86 calling convention, that the EBP pointer, or register, actually happens to point to something on the stack which is going to be called the saved EBP. It's a separate location, sort of after all the variables but before the return address. And this is the thing that's being saved by those first couple of instructions at the top.

And you actually sort of examine it. In GDB you can say, examine x, some value, so the EBP pointer value. So that's the location of the stack, D0B8. Indeed, it's actually above even the i variable. So it's great. And it has some other value that happens to be the EBP before this function was called. But then, sort of one more memory location up is going to be the return address. So if we print EBP plus four, there's something else there, this 0x08048E5F. And let's actually see where that's pointing.

So this is something you're going to do a lot in the lab. So you can take this address. And you can try to disassemble it. So what is this guy? Where did we end up? So GDB actually helpfully figures out which function contains that address. So 5F. This is the guy that our return address is pointing to. And as you can see, this is the instruction right after the call to redirect. So when we return from redirect, this is exactly where we're going to jump and continue execution. This is, hopefully, fairly straightforward stuff from double oh four, some standard OS class.

OK. So where are we now? Just to recap, we can try to disassemble our instruction pointer. So we're at the beginning of redirect right now. And we can run for a bit, and maybe run that getS() function. So OK, we run next. What this does is it runs getS() and it's waiting for getS() to return. We can provide our bad input to getS() and try to get it to crash again and see what's going on, really, there, right? So we can paste a bunch of As again.

OK. So we got out of getS() and things are actually still OK, right? The program is still running. But we can try to figure out, what is in memory right now and why are things going to go wrong? Actually, what do you guys think? What happened, right? So I printed out a bunch of As. What did getS() do to the memory? Yeah, yeah. So it just keeps writing As here, right? All we actually passed to getS() was a single pointer, the start of this address, right? So this is the argument to getS(), is a pointer to this memory location on the stack. So it just kept writing As. And it doesn't actually know what the length is, so it just keeps going, right? It's going to override As all the way up the stack, past the return address, probably, and into whatever was up the stack above us.

So we can check whether that's the case. So we can actually print the buffer. And in fact, it tells us, yeah, we have 180 As there, even though the buffer should be 128 elements large. So this is not so great. And we can actually, again, examine what's going on in that EBP pointer. Dollar sign, EBP. So in fact, yeah. It's all 0x41, which is the ASCII encoding of the letter A. And in fact, the return address is probably going to be the same way, right? If we print the return address, it's also all As. That's not so great.

In fact, what's going to happen if we return now is the program will jump to that address, 41414141. And there's nothing there. And it'll crash. That's the segmentation fault you're getting. So let's just step up to it and see what happens. So let's run next. So we keep stepping through the program. And we can see where we are. OK. We're getting close to the end of the function. So we can step over two more instructions. nexti. And now we can disassemble again. OK.

We're now just at the return instruction from this function. And we can actually figure out. So as you can see, at the end of the function, it runs this leave x86 instruction, which basically restores the stack back to where it was. So it sort of pushes the stack pointer all the way back to the return address using the same EBP. That's what it's basically for. And now, the stack is pointing at the return address that we're going to use. And in fact, it's all A's. And if we run one more instruction, the CPU is going to jump to that exact memory address and start executing code there and crash, because it's not a valid address that's in the page table.

So let's actually see, just to double check, what's going on. Let's print our buffer again. Our buffer-- well, that's actually kind of interesting, right? So now, buffer, for some reason it only says A repeats 128 times. Whereas if you remember before, it said A repeated 180 times in our buffer. So what happened? Yeah?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah, yeah. Exactly. So there's actually something going on after the buffer overflow happens that changes what's going on. So actually, if you remember, we do this A to i conversion of the string to an integer. And if you provide all As, it actually writes zero to this memory location. So a zero, if you remember, terminates strings in C. So GDB now thinks, yep, we have a perfectly well-terminated 128 byte string of all As.

But you know, it doesn't really matter, because we still have those As up top that already corrupted our stack. OK. That was actually kind of an important lesson that-- it's actually a little bit tricky, sometimes, to explore these buffer overflows because, even though you've already changed lots of stuff on the stack, you still have to get to the point where you use the value that you have somehow placed on the stack. So there's other code that's going to run after you've managed to overflow some buffer and corrupt memory.

You have to make sure that code doesn't do something silly like, if it's A to i, just exited right away, as soon as it saw a non-integer value, we might not get to jump to all this 41414141 address. So you have to massage your input in some cases. Maybe not so much in this case. But in other situations, you'll have to be careful in constructing this input.

OK, so just to see what happens, we can jump one more time. Well, let's look at our register. So right now, our EIP, the sort of instruction pointer, is pointing at the last thing in redirect. And if we step one more time, hopefully we'll jump to, finally, that unfortunate 4141 address. Over here. And in fact, yep. The program now seems to be executing there. If we ask GDB to print the current set of registers, yep, the current instruction pointer is this strange value. And if we exclude one more instruction, it's going to crash because that's finally trying to execute an instruction pointer that doesn't correspond to a valid page in the operating system's page table for this process. Make sense? Any questions?

All right. Well, I've got a question for you guys, actually. So what happens-- you know, it seems to be exploitable. Or well, OK. Maybe let's first figure out why this is particularly bad, right? So why is it a problem? So not only does our program crash, but presumably we're going to take it over. So I guess, first simple question is, OK, so what's the problem? What can you do? Yeah?

AUDIENCE: You can do whatever you want.

PROFESSOR: Yeah. So I was actually pretty silly and just put in lots of As. But if you were careful about knowing where to put what values, you might be able to put in a different value and get it to jump somewhere else. So let's see if we can actually do this, right? We can retrace this whole thing. OK. Re-run the program again. And I guess I have to reset the breakpoint. So I can break and redirect again. And run. And this time, I'll, again, next, supply lots of As and overflow things. But I'm not going to try to carefully construct-- you know, figure out which point in these As corresponds to the location in the stack. That's something you guys are going to have to do for lab one.

But suppose that I overflow the stack here. And then I'm going to manually try to change things on the stack to get it to jump to some point I want to jump to. And in this program, OK, so let's again-- nexti. Where are we? We're at, again, at the very end of redirect. And let's actually look at the stack, right?

So if we examine esp here, we see our corrupted pointer. OK. Where could we jump to? What interesting things could we do? Unfortunately, this program is pretty limited. There's almost nothing in the program's code where you could jump and do anything interesting. But maybe we can do a little bit of something interesting. Maybe we'll find the printf in main and jump directly there, and get it to print the x value, or x equals something. So we can do this. We can actually disassemble the main function. And main does a bunch of stuff, you know, initializes, calls redirect, does some more stuff, and then calls printf.

So how about we jump to this point, which is, it sets up the argument to printf, which is x equals percent d, and then actually calls printf. So we can actually take this value and try to stick it in the stack. And should be able to do this with the debugger pretty easily, at least. So you can do this set {int} esp equals this value. So we can examine esp again and, indeed, it actually has this value. So if we continue now, well, it printed out x equals some garbage, which I guess happens to be just whatever is on the stack that was passed to printf. We didn't correctly set up all the arguments because we jumped in the middle of this calling sequence. But yeah, we printed this value. And then it crashed.

Why did crash? Why do you think? What actually happens, right? So we jump to printf. And then, something went wrong. Yeah? Well, we changed the return address so that when we return from redirect, we now jump to this new address, which is that point up there, right after printf. So where's this crash coming from? Yeah?

AUDIENCE: Is it restricted because your i is supposed to be some sort of integer, but--

PROFESSOR: No, actually, well the i is like, well it's a 32-bit register. So whatever's in the register, it'll print. In fact, that's the thing that's in the register. So that's OK. Yeah?

AUDIENCE: [INAUDIBLE] main returns.

PROFESSOR: Yes. Actually, yeah. What's going on is, you have to sort of-- OK, so this is the point where we jumped. It's set up some arguments. It actually calls printf. printf seems to work. printf is going to return. Now actually, that's fine, because this call instruction put a return address on the stack for printf to use. That's fine. Then main is going to continue running. It's going to run the sleeve instruction, which doesn't do anything interesting. And then it does another return. But the thing in this-- up to the stack, it doesn't actually have a valid return address. So presumably, we return to some other who knows what memory location that's up on the stack and jump somewhere else.

So unfortunately, here, our pseudoattack didn't really work. It ran some code. But then it crashed. That's probably not something you want to do. So if you really wanted to be careful, you would carefully plant not just this return address up on the stack, but maybe you'd figure out, where is this second red going to get its return address from, and try to carefully place something else on the stack there that will ensure that your program cleanly exits after it gets exploited so that no one notices. So this is all stuff you'll sort of try to do in lab one in a little bit more detail.

But I guess one thing we can try to think about now is, we sort of understand why it's bad to jump to the-- or to have these buffer overflows. One problem, or one sort of way to think of this is that, the problem is just because the return address is up there, right? So the buffer keeps growing and eventually runs over the return address. What if we flip the stack around? You know, some machines actually have stacks that grow up.

So an alternative design we could sort of imagine is one where the stack starts at the bottom and keeps going up instead of going down. So then, if you overflow this buffer, you'll just keep going up on the stack, and maybe there's nothing bad that will happen. Yeah?

AUDIENCE: [INAUDIBLE].

PROFESSOR: So you're right. It might be that, if you have-- well, so let me draw this new stack diagram. And we'll sort of try to figure out what it applies to and not. But OK. So we'll basically just invert the picture. So when you call redirect on this alternative architecture, what's going to happen is the return address is going to go here on the stack. Then we'll have our i variable, or maybe the saved EBP. Then we'll have our i variable. And then we'll have buff. So we'll have buff of zero, buff 127, and so on, right? So then when we do the overflow, it overflows up there and maybe doesn't hit anything bad.

I guess what you're saying is that, well, maybe we had a buffer down there. And if we had a buffer down there, then yeah, that seems kind of unfortunate. It could overrun this return address. So you're right. So you could still run into problems on this stack growing up.

But what about this exact program? Is this particular program safe on machines where the stack grows up? So just to recap what the program read is this guy. Yeah?

AUDIENCE: Still going to overwrite [INAUDIBLE] as a return value.

PROFESSOR: Yeah. So that's actually clever, right? So this is the stack frame for redirect. I guess it actually spans all the way up here. But what actually happens when you call getS() is that redirect makes a function call. It actually saves its return address up here on the stack. And then getS() starts running. And getS() puts its own saved EBP up here. And getS() is going to post its own variables higher up. And then getS() is going to fill in the buffer.

So this is still problematic. Basically, the buffer is surrounded by return initials on all sides. Either way, you're going to be able to overflow something. So at what point-- suppose we had a stack growing up machine. At what point would you be able to take control of the program's execution then? Yes, and that is actually even easier in some ways. You don't have to wait until redirect returns. And maybe there was like, stuff that was going to mess you up like this A to i. No. It's actually easier, because getS() is going to overflow the buffer. It's going to change the return address and then immediately return and immediately jump to wherever you sort of tried to construct, makes sense.

So what happens if we have a program like this that's pretty boring? There's like no real interesting code to jump to. All you can do is get it to print different x values here. What if you want to do something interesting that you didn't-- yeah?

AUDIENCE: I mean, if you have an extra cable stack, you could put arbitrary code that, for example, executes a shell?

PROFESSOR: Yeah yeah yeah. So that's kind of clever, right, because you actually can supply other inputs, right? So at least, well-- there's some defenses against this. And we'll go over these in subsequent lectures. But in principle, you could have the return address here that you override on either the stack up or stack down machine. And instead of pointing it to some existing code, like the printf inside of main, we can actually have the return address point into the buffer. So it's previously just some location on the stack.

But you could jump there and treat it as executable. So as part of your request, you'll actually send some bytes of data to the server, and then have the return address or the thing you overwrite here point to the base of the buffer, and you'll just keep going from there. So then you'll be able to sort of provide the code you want to run, jump to it, and get the server to run it.

And in fact, traditionally, in Unix systems, what adversaries would often do is just ask the operating system to execute the binsh command, which lets you sort of type in arbitrary shell commands after that. So as a result, this thing, this piece of code you inject into this buffer, was often called, sort of for historical reasons, shell code. And you'll try to construct some in this lab one as well. All right. Make sense, what you can do here? Any questions? Yeah?

AUDIENCE: Is there a separation between code and data?

PROFESSOR: Right. So is there a separation between code and data here? At least, well, historically, many machines didn't enforce any separation of code and data. You'd just have a flat memory address space. The stack pointer points somewhere. The code pointer points somewhere else. And you just execute wherever the code pointer, instruction pointer is pointing. Modern machines try to provide some defenses for these kinds of attacks. And what modern machines often do is, they actually associate permissions with various memory regions. And one of the permissions is execute.

So the part of your 32-bit or 64-bit address space that contains code has the execute permission. So if your instruction pointer points there, the CPU will actually run those things. And the stack and other data portions of your address space typically don't have the execute permission. So if you happen to somehow set your instruction pointer to some non-code memory location, you can set it, but the CPU will refuse to execute it. So this is a reasonably nice way to defend against these kinds of attacks. But it doesn't prevent quite everything.

So just a question. OK. So how would you bypass this if you had this non-executable stack? You actually saw this example earlier, right, when I actually jumped to the middle of main. So that was a way of sort of exploiting this buffer overflow without having to inject new code of my own. So even if the stack was non-executable, I would still be able to jump in the middle of main. In this particular case, it's kind of boring. It just prints x and crashes.

But in other situations, you might have other pieces of code in your program that are doing interesting stuff that you really do want to execute. And that's sort of called return to libc attacks for, again, somewhat historical reasons. But it is a way to bypass the security measures. So in the context of buffer overflows, there's not really a clear cut solution that provides perfect protection against these mistakes because, at the end of the day, the programmer did make some mistake in writing this source code. And the best way to fix it is probably just to change the source code and make sure you don't call getS() very much, like the compiler warned you. And there's more subtle things that the compiler doesn't warn you about. And you still have to avoid making those calls.

But because it's hard, in practice, to change all the software out there, many people try to devise techniques that make it more difficult to exploit these bugs. For example, making the stack non-executable, so you can't inject the shell code onto the stack, and you have to do something slightly more elaborate. And next couple of lectures, next two lectures, actually, we'll look at these defense techniques. They're not all perfect. But they do, in practice, make it much more difficult for that hacker to exploit things. Question?

AUDIENCE: I just have a general administrative question.

PROFESSOR: Yeah?

AUDIENCE: I was wondering if there was a final? And also if there are quizzes, and what dates--

PROFESSOR: Oh yeah. Yeah, I think if you go to the schedule page, there's two quizzes. And there's no final during the final week, but there's a quiz right before it. So you're free for the final week, but there's still something at the end of the class. Yeah. All right. OK.

So I think that's probably it for buffer overflows. I guess the one question is, so what do you do about mechanism problems? And the general answer is to probably have fewer mechanisms. So as we saw here, if you're relying on every piece of software to enforce your security policy, you'll inevitably have mistakes that allow an adversary to bypass your mechanism to exploit some bug in the web server. And a much better design, and one but you will explore in lab two, is one where you structure your whole system so the security of the system doesn't depend on all the pieces of software enforcing your security policy.

The security policy is going to be enforced by a small number of components. And the rest of the stuff actually doesn't matter, for security purposes, if it's right or wrong. It's not going to violate your security policy at all. So this, kind of minimizing your trusted computing base is a pretty powerful technique to get around these mechanism bugs and problems that we've looked at today, at least in a little bit of detail.

All right. So read the paper for Monday. And come to Monday's lecture. And submit the questions on the website. See you guys then.