PROFESSOR: This is very exciting. In the previous lecture, we learned all about buffer overflow attacks, and today we're going to continue to discuss some techniques to launch these attacks. So, the basic idea of all these buffer overflow attacks is as follows. So, first of all, they leverage a couple different facts.

So, one thing that they leverage is that system software is often written in C.

And so by system software, I mean things like databases, compilers, network servers, things like that. And you can also think of things like your favorite command shell. All of those types of things are typically written in C. So, why are these things typically written in C? Well, they're written in C because our community, of course, is obsessed with speed. And so C is supposed to be like high-level assembly, it takes us very close to the hardware, and so as a result, all these very mission critical systems are written in this very low level language. Now, the problem with things being written in C is that C actually exposes raw memory Addresses.

Right? And so not only does it expose raw memory addresses, but it also performs no bounds checking when programs manipulate those raw addresses. Right? And so as you can imagine, this is a recipe for disaster. OK? So, once again, why doesn't C check these bounds? Well, one reason is because the hardware doesn't do that. And people who write in C typically want the max amount of speed possible. The other reason is that in C, as we'll discuss later, it can actually be very difficult to determine the semantics of what it means to have a pointer that's actually in bounds. So, in some cases, it would be very difficult for the C runtime to automatically do that. Now we'll discuss some techniques which will actually try to do that type of automatic inference. But as we'll see, none of these techniques are fully bulletproof.

And so these attacks also leverage knowledge of the x86 architecture. And by knowledge of that architecture I mean things like what's the direction that the stack grows, right? What are the calling conventions for functions? When you invoke a C function, what is the stack going to look like? And when you allocate an object on the heap, what are those chief allocation structures going to look like? And so let's look at a simple example. It's very similar to something that you saw in the last lecture. So, we've got you're standard read request up here. And then you've got a buffer. That's here.

And by now you've probably trained your lizard brain instincts-- whenever you see a buffer you're probably filled with fear-- that is the right attitude. And so we've got the buffer up here, and then we've got the canonical int i. And then we've got the infamous "gets" command. And then you've got some other stuff here. Right?

So as we discussed in lecture last week, this is problematic, right? Because this gets operation here does not actually check the bounds on the buffer. So, what can happen is that if the user actually supplies the buffer-- and actually put that guy up here, for example-- if that buffer comes in from the user and we use this unsafe function here, we can actually overflow this buffer. We can actually rewrite stuff that's on the stack. So, just a reminder of what that stuff looks like-- let's look at a stack diagram here-- so let's say here we've got I. Let's say here we've got a buf

Right? So, we've got the first address of buffer here. We've got the last one up here. I apologize for my handwriting, I'm used to writing on the marker board. You should pray for me. So, anyways, then up here, we've got the saved value of the break pointer. We've got the return address for the function there. And then we've got some other stuff from the previous frame. So, don't forget, we've got the stack pointer, which goes there. And then we've got the new break pointer, which goes here.

The entry stack pointer goes there, and then somewhere up here, we've got the entry break point. Right? So, just as a reminder, the way that the Stack Overflow works is that basically, it goes this way. Right? So, when the gets operation is called, we start writing bytes into buf, and eventually it's going to start overwriting these things that are on the stack. And so this is basically-- should all look pretty familiar to you. So. What does the attacker do to take advantage of that? Basically supplies that long input. And so the key idea here is that this can be attacker-supplied.

And so, if this return address is attacker-supplied, then basically the attacker can determine where this function's going to jump to after [INAUDIBLE] execution. So, what can the attacker do once it's actually been able to hijack that return address, and jump wherever it wants. Well, basically the attacker is now running code with the privileges of the process that it's just hijacked, for example. So, if that process was a high priority process, let's say it was running root, or admin, whatever they call the super-user of your favorite operating system, then now, that program, which is controlled by the attacker, can do whatever it wants using the authority of that high-priority program.

So, it can do things, like it could maybe read files, it can send spam, let's say if you corrupted a mail server. It can even do things like actually defeat firewalls, right, because the idea of a firewall is that there's going to be this distinction between good machines that are behind the firewall and bad machines that are outside of the firewall. So, typically machines are inside of the firewall, they have a lot of trust with each other. But if you can subvert a machine that is actually inside the firewall, right, that's great. Because now you can just sort of skip past a lot of those checks that those machines don't have because they think that you're a trusted individual.

So, one thing you might be thinking, or I remember I was thinking this when I was a student, was, "OK, fine, so I've showed you how to do this buffer overflow, but why didn't the OS stop this? Right? Isn't the OS supposed to be that thing that's sort of sitting around like Guardians of the Galaxy and defending all this kind of evil stuff from happening?" The thing to note is that the OS actually isn't watching you all the time. Right? The hardware is watching all the time. It's the thing that's actually fetching instructions, and decoding them, and doing things like that.

But to a first approximation, what does the OS do? It basically sets up some page table stuff, and then it basically lets you, the application, run, and if you ask the operating system for services-- so for example, you want to send a network packet, or you want to do some IPC, or things like that, then you'll invoke a system call, and you'll actually trap into OS. But other than that, the operating system is not looking at each and every instruction that your application is executing. So, in other words, when this buffer overflowed, it's not like the operating system was looking at each of these memory axises for signs that [INAUDIBLE]. Right? All of this address space belongs to you, this [INAUDIBLE] process right, so you get to do with it what you want to do with it, right? Or at least this is the whole C attitude towards life, right? Life fast, die young.

So. That's why the operating system can't help you right there. So, later in the lecture, we will discuss some things that the operating system can do with respect to the hardware so that it can help protect against these types of attacks. Once again, it's actually just the hardware that's interposing on every little thing that you do. So, you can actually take advantage of some of that stuff, for example, using special types of [INAUDIBLE] protections and things like that, that we'll discuss a little bit later.

That's basically an overview of what the buffer overflow looks like. So, how are we gonna fix these things? So, one fix for avoiding buffer overflow is to simply avoid bugs in your C code. This has the nice advantage of being correct by construction, right. If you don't have any bugs in your program, ipso facto the attacker cannot take advantage of any bugs. That's on the professor, I get paid to think about something deeply like that. Now, this of course, is easier said than done. Right?

There's a couple of very straightforward things that programmers can do to practice good security hygiene. So, for example, functions like this gets function, right? These are kind of like go-tos, these are now known to be bad ideas. Right? So, when you compile your code, and you include functions like this-- if you're using a modern compiler, GCC, Visual Studio, whatever, it will actually complain about that.

It'll say, hey, you're one of these unsafe functions. Consider using [? FGADS ?], or using a version of [INAUDIBLE] that actually can track the bounds of things. So, that's one simple thing that programmers can do.

But note that a lot of applications actually manipulate buffers without necessarily calling one of these functions. Right? This is very common in network servers, things like that. They'll define their own custom parsing routines, then make sure that things are extracted from the buffers in the way that they want. So, just restricting yourself to these types of things won't solve the problem completely. So, another thing that makes this approach difficult is that it's not always obvious what is a bug in a C program.

So, if you've ever worked on a very large scale system that's been written in C, you'll know that it can be tricky if you've got some function definition that takes then 18 void star pointers. I mean, only Zeus knows what all those things mean, right? And so it's much more difficult in a language like C, that has weak typing and things like that, to actually understand as a programmer what it means to have a bug, and what it means to not have a bug. OK?

So, in general, one of the main themes that you'll see in this class is that C is probably the spawn of the devil, right? And we use it because, once again, we typically want to be to be fast, right? But as hardware gets faster and as we get more and better languages to write large-scale systems code, we'll see that maybe it doesn't always make sense to write your stuff in C. Even if you think it has to be fast. So, we'll discuss some of that later and later lectures.

So, that's one approach, avoiding bugs in the first place. So, another approach-- is to build tools that allow programmers to find bugs. And so an example of this is something that's called static analysis. Now we'll talk a little bit more about static analysis in later lectures, but suffice it to say that static analysis is a way of analyzing the source code of your program before it even runs and looking for potential problems. So, imagine that you have a function like this. So, the [INAUDIBLE] foo function, it takes in a pointer. Let's say it declares an integer offset value. It declares another pointer and adds the offset to that pointer. Now, even just at this moment in the code, right, static analysis can tell you that this offset variable is un-initialized. Right?

So, essentially you can do things like saying, is there any way, is there any control floating through this program by which offset could have been initialized before it was actually used this in this calculation here. Now, in this example it is very simple to see the answer is no. Right? You can imagine that if there were more branches, or things like this, it would be more difficult to tell. But one thing that a static analysis tool can tell you, and in fact, one thing that [? popular ?] compilers will tell you, is you'll compile this, and it'll say, hey buddy, this has not been initialized. Are you sure, is this what you want to do?

So, that's one very simple example of static analysis. Another example of what you can do is, let's say after this, we have a branch condition here. Right? So, you say, if the offset is greater than eight, then we'll call some function bar, and passing the offset. Now, one thing you can note about this is that this branch condition here actually tells us something about what the value of offset is. Right? Ignoring the fact that it wasn't initialized , we do know that once we get here, we know the offset actually has to be greater than eight. So, in some cases, what we can do is actually propagate that constraint, that notion that the offset must be greater than eight, into our analysis of bar. Right? So, when we start statically analyzing bar, we know that offset can only take certain values. So, once again, this is a very high-level introduction to static analysis, and we'll discuss it more in later lectures.

But this is a basic intuition of how we might be able to detect some types of bugs without even executing your code. So, does that all makes sense? So, another thing you can think about doing too is what they call program fuzzing. So, the idea behind program fuzzing is that essentially you take all of the functions in your code, and then essentially throw random values for input to those functions. And so the idea is that you want to have high code coverage for all of your tests. So, if you go out in the real world, typically when you check in unit test, you can't just do things like, I tried values two, four, eight, and 15, because 15 is an odd number, so I probably tested all the branches right.

What you actually have to do is you have to look at things like, like I said how many branches in the program overall were actually touched by your test code, right? Because that's typically where the bugs hide. The programmers don't think about the corner cases, and so as a result, they do have some unit tests that pass. They even have bigger tests that pass. But they're not actually pinning all the corner cases in the program. So, static analysis can actually help with this fuzzing here. Once again, using things like this notion of constraint. So, for example, in this program here, we have this branch condition here that specified the offset being greater than eight.

So, we can know what that offset is statically. So, we can make sure that if we're automatically generating fuzzed inputs, we can ensure that one of those inputs hopefully will ensure that, somehow, offset is less than eight, one will ensure that offset's equal to eight, one will ensure that it's greater than eight. So, does that all make sense? Cool.

So, that's the basic idea behind the notion of building tools to help programmers find bugs. So, the nice thing is that even partial analysis can be very, very useful, particularly when you're dealing with C. A lot of these tools that we'll discuss, to prevent against things like buffer overflow or initialized variables, they can't catch all the problems. Right? But they can actually give us forward progress towards making these programs more secure. Now, of course, the disadvantage of these things is that they're not complete. Forward progress is not complete progress. And so it's still a very active area of research of how you defend against security exploits in C and just in programs in general.

So, those were two approaches to deal with defending against buffer overflow. There's actually some other approaches. So, a third approach you might think about using is the use [INAUDIBLE]. And so examples of these are things like Python, Java, C#-- I'm not going to put up Pearl there because people who use Pearl are bad people.

So you can use a memory-safe language like that. And this is to a certain extent seems like the most obvious thing that you could do. I just told you over there that basically C is high-level assembly code, and it exposes raw pointers and does all these things that you don't want it to do, and it doesn't do things you do want it to do, like [INAUDIBLE]. So, why not just use one of these high level languages?

Well, there's a couple reasons for that. So, first of all, there's actually a lot of legacy code that's out there. Right? So, it's all fine and dandy if you want go out and start your new project and you want to write it in one of these really safe languages.

But what if you've been given this big binary or this big source code distribution that's been written in C, it's been maintained for 10, 15 years, it's been this generational project, I mean our children's children will be working on it. You can't just say, I'm just going to write everything in C# and change the world. Right? And this isn't just a problem in C, for example. There's actually systems that you use that you should be afraid, because they actually use Fortran and COBOL code. What? That's stuff from the Civil War.

So, why does that happen? Once again, the reason why it happens is because as engineers, we kind of want to think, oh, we can just build everything ourselves, it'll be awesome, it'll be just the way that I want it, I'll call my variables the things that I want. When in world, that doesn't happen. Right? You show up on your job, and you have this thing that exists, and you look at the code base, and you say, well, why doesn't it do this? And then you say, listen. We'll deal with that in V2.

But for now, you got to make things work because the customers are taking away their money. So, there's basically this huge issue of legacy code here, and how do we deal with it? And as you'll see the with the baggy bounds system, One of the advantages of it is that it actually inter-operates quite well with this legacy code. So, anyway, this is one reason why you can't just necessarily make all these buffer overflow problems go away by using one of these memory-safe languages.

So, another challenges is that what if you need low-level access to hardware? This might happen if you're writing something like a device driver or something like that.

So, in that case, you really do need that the benefits that C gives you in terms of being able to look at registers and actually understand a little of [INAUDIBLE] and things like that. There's another thing too, which people always bring up and which I've alluded to before, but it's performance. Right? So, if you care about performance, typically the thing that you're told is you've got to write in C, otherwise you're just going to be so slow, you're going to get laughed out of code academy or whatever.

Now, this is increasingly less of an issue. Like the perf stuff. Because people have actually gotten very good with doing things like making better compilers that have all kinds of powerful optimizations. And also, there are these things called Gits which actually really reduce the cost of using these memory-safe languages. So, have you guys heard of Gits before? So, I'll give you a very brief introduction to what it is. The idea is that, think about a language like Java, or JavaScript. It's very high level, it's dynamically tight, right, it has automatic heat management, things like that. So, typically, when these languages first came out, they were always interpreted. Right? And by interpreted I mean they didn't actually execute raw x86 instructions.

Instead, these languages were compiled down to some type of intermediate form. You may have heard of things like the JVM, the Java Virtual Machine byte code, things like that. Right? You basically had a program that sat in a loop and took these byte codes, and basically executed the high level instruction that was encoded in that byte code. So, for example, some of the JVM byte codes dealt with things like pushing and popping things up on the stack. So, you have a program that would go through a loop, operate that stack, and simulate those operations. OK. So, that all seemed fine and dandy, but once again, all of the speed freaks out there were like, what about the perf? This too slow. You've got sort of that interpreter sitting in that loop, and getting in the way of our bare metal performance.

So, what people started to do is actually take these high level interpreter languages and dynamically generate X86 code for them on the fly. Right? So, in terms of just in time compilation, that means I take your snippet of JavaScript, I take your snippet of Java whatever, and I actually spend a little bit of time upfront to create actual raw machine instructions. Raw x86 that will run directly on the bare metal. So, I take that initial performance hit for the Git compilation, but then after that, my program actually does run on the raw hard drive. Right? So, things like the perf argument are not necessarily as compelling as they used to be, because of stuff like this.

There's also some crazy stuff out there, like ASN.js. So, we can talk more about this offline if you actually are a JavaScript packer. But there are actually some neat tricks that you can do, like compiling down JavaScript to very restricted subset of the language that only operates on arrays. Right, so what this allows you to do is actually get rid of a lot of the dynamic typing overhead in standard JavaScript, and you can actually get JavaScript code now to run within 2x of raw C or C++ performance. 2x might sound like a lot, but it used to be things like 10x or 20z. So, we're actually making a lot of progress on that front.

And so the other thing to keep in mind with performance, too, is that a lot of times, you don't need performance as much you might think that you do. Right? So, think about it like this. Let's say that your program is actually IO bound. So, it's not CPU bound. In other words, let's say that your program spends most of its time waiting for network input, waiting for disk input, waiting for user input, things like that. In those types of cases, you don't actually need to have blazing fast raw compute speed. Right? Because your program actually isn't spending a lot of time doing that kind of stuff.

So, once again, this perf argument here, you've got to take this stuff with a grain of salt. And I actually see a lot students who struggle with this. So, for example, I'll ask someone to go out and write me a very simple program to parse a text file. So, they spend all this time trying to get this to work in C or C++ and it's super fast and uses the templates and all that kind of stuff. But it's like a one line solution in Python. And it essentially runs just as fast. And you could develop it much, much easier. So, you just have to take these perf arguments with a grain of salt.

So, anyway, we've discussed the three ways you can possibly avoid buffer overflow. So, just avoid bugs in the first place. LOL, that's difficult to do. Approach two, you can build tools to help you discover those bugs. Then approach three is, in a certain sense, you can push those tools into the runtime. You can actually hopefully rely on some of their language runtime features to prevent you from seeing raw memory addresses. And you can do things like balance checking, and so on and so forth. Once again, as we discussed before, there's a lot of legacy C and C++ code out there. So, it's difficult to apply some of these techniques, particularly number two and number three, if you've got to deal with that legacy code.

So, how can we do buffer overflow mitigation despite all these challenges? Besides just, you know, dropping out of computer science classes and becoming a painter, or something like that. So, what actually is going on in a buffer overflow? So, in a buffer overflow the attacker exploits two things. So, the first thing that the attack is going to exploit is gaining control over the instruction pointer. Right? And by this, I mean that somehow, the attacker figures out someplace in the code that it can make the program jump to against the program's will. Now, this is necessary but insufficient for an attack typically to happen. Because the other thing that the attacker needs to do is basically make that pointer point to malicious code.

Right? So, how are we going to basically make the hijacked IP, instruction pointer, point to something that does something useful for the attacker. So, what's interesting is that in many cases, it's often fairly straightforward for the attacker to put some interesting code in a memory. So we looked at some of those shell code attacks in the last lecture, where you can actually embed that attack code in a string. As we'll discuss a little bit today and more in the next lecture, you can actually take advantage of some of the pre-existing code the application has and jump to in an unexpected way to make some evil things happen.

So, typically, figuring out what code the attacker wants to run, maybe that's not as challenging as actually being able to force the program to jump to that location in memory. And the reason why that's tricky is because, basically, the attacker has to know in some way where it should jump to. Right? So, as we'll see in a second, and as you actually saw in the last lecture, a lot of these shell code attacks actually take advantage of these hard-coded locations in memory where the instruction pointer needs to get sent to. So, some of the defenses that we're about to look at can actually randomize things in terms of code layout, heap layout, and make it a little difficult for the attacker to figure out where things are located.

So, let's look at one simple mitigation approach first. So, this is the idea of stack canaries. So, the basic idea behind stack canaries is that, during a buffer overflow, it's actually OK if we allow the attacker to overwrite the return address if we can actually catch that overwrite before we actually jump to the place that the attacker wants us to go. So, basically, here's how it works. Let's return to Neal stack diagram. Essentially we have to think of it as a magic value. Basically, in front of the return address. Such that any overflow would have to hit the canary first, and then hit the return address. And if we can check that canary before we return from the function, then we can detect the evil.

So, let's say that, once again, we've got the buffer here. Then we're going to put the canary here. And then this will be the save value of the break pointer. And then this will be the return address. So, once again, remember the overflow goes this way. So the idea is that if the overflow wants to get to that return address, it first has to trample on this canary thing here, right? You have a question?

AUDIENCE: Why does it have to touch the canary?

PROFESSOR: Well, because-- assuming that the attacker doesn't know how to jump around in memory arbitrarily-- the way that tradionally [INAUDIBLE] overflow attacks work is that you look in GB, figure out where all this stuff is. And then, you essentially have this string, [INAUDIBLE] radius grows this way.

Now, you're correct that if the attacker could just go here directly, then all the bets are off. But in the very simple overflow approach, everything just has to grow strictly that way.

So the basic idea behind the canary is that we allow the buffer overflow exploit to take place. But then we have run time code that, at the time of the return from the function, is going to check this canary and make sure that it has the right value.

Right? So it's called the canary because back in the days, when PETA wasn't around, you could use birds to test for evil things. So that's why it's called canary.

AUDIENCE: My question is if the attacker is able to overwrite the return address, and modify the canary, how does he check that the canary was not modified, but was going to be performed? So the attacker overwrites the return address, right? So how is the check that the canary was modified--

PROFESSOR: Yeah. So basically, you have to have some piece of code that will actually check this before the return takes place. So in other words, you're right. There has to be that order in there.

So essentially, what you have to do is you have to have a support from the compiler here that will actually extend the calling convention, if you will. Such that part of the return sequence is before we actually treat this value as valid, make sure this guy hasn't been trampled. Then, and only then, can we think of going somewhere else.

AUDIENCE: I think I might be jumping the gun here, but doesn't this assume that the attacker can't find out or guess what the canary value is?

PROFESSOR: That, in fact, is the very next thing my lecture is. If I had prizes, you'd get one. I don't have any. But good for you. Gold star.

That's exactly correct. So one of the next things I'd like to say is what's the problem with this scheme? What if, for example, on every program, we always put the value a? Just like four values of a. So this is like a single [INAUDIBLE] at work, right?

Then you'd have that exact problem that you just mentioned. Because then, the attacker-- this gets back to your question-- he or she knows how big this is. This is deterministic on every system.

So you just make sure that your buffer overflow has a bunch of a's here, and then you overwrite this side. So you're exactly right about that. And so there's basically different types of values you could put between this canary to try to prevent that.

One thing that you can do here is you can use-- this is sort of a very funny type of canary, but it basically exploits the ways that a lot of C progams and C functions handle special characters. So imagine that you used this value for the canary. So the binary value is 0, which is like the null byte, the null character in ASCII. Carriage return line feed, and then the negative 1.

What's funny about this is that a lot of the functions that you can exploit-- that manipulate strings, for example-- they will stop when they encounter one of these words, or one of these values. So you can imagine that you're using some string manipulation function to go up this way. It's going to hit that null character. Oops-- it's going to stop processing. Right?

Or maybe if you're using a line-oriented function-- carriage return, line feed-- that's often used as the line terminator. So once again, you're using that dangerous function that's trying to go this way. It hits that. Oops, it's going to quit.

And the negative 1 is another similar magic token. So that's one way you can get around that. One second.

And then another thing you can do is you can use a randomized value. So here, you just [INAUDIBLE] from this whole idea of trying to figure out what exactly it is that might cause that attack to terminate. And you just pull some random number and either make it difficult for the attacker to guess what that is.

Now, of course, this random value-- its strength is basically based on how difficult it is for the attacker to guess that. So the attacker, for example, can understand that if there's only, let's say, three bits of entropy in your system, then maybe the attacker could use some type of forced attack, so on and so forth. So one thing to keep in mind, in general, is that whenever someone tells you, here's a randomized offense against attack foo, if there are not a lot of random bits there, that attack may not give you as much defense as you think. You had a question?

AUDIENCE: Usually what tends to happen is you read from another buffer and you write into that buffer on the stack. So in that situation, it seems like that promiscuous canary is kind of useless. Because if I read from the [INAUDIBLE], I know what the canary is.

And I have this other buffer that I control. And I never check. And in that buffer, I can put as much of it as I want. I don't want the promiscuous canary, so I can overwrite it very safely. So I don't see how this really works, and in what scenario it's-- you're assuming you're reading from the buffer on this stack and you're going to stop--

PROFESSOR: Well, we're assuming-- we're writing to the buffer. So basically, the idea is that you write some [? two-long ?] string this way. And then the idea is that if you can't guess what this is, then you can't, basically, put this value inside of your overflow string.

AUDIENCE: But you said it's deterministic, right? 0, CR, LF, negative 1.

PROFESSOR: Oh, yeah. Right. OK. So I think I understand your question now. Yes. If you use this system here, with the deterministic canary, and you essentially are not using one of these functions from, let's say, the standard library that would be fooled by this, then, yeah, you can defeat the system that way.

AUDIENCE: But I can use string CPIs and the destination can be buffered. And the source can be [INAUDIBLE]. And that would not protect me against that.

PROFESSOR: I'm not sure I understand the attack, so.

AUDIENCE: So the string CPI would take home the user input for my data, would overwrite canary-- oh, and you're saying-- hmm, actually, I understand what you're saying.

PROFESSOR: Right? Because the idea is that you can fill this buffer with bytes from wherever, right? But the idea is that unless you can guess this, then it doesn't matter. But you're correct. In general, anything that allows you to guess this or randomly get that value correct will lead to the feed of the system.

AUDIENCE: In terms of [INAUDIBLE], can you just take something like the number of seconds or milliseconds since the epoch and use that at the [INAUDIBLE]?

PROFESSOR: Well, as it turns out, a lot of times, calls that get [INAUDIBLE] don't contain as much randomness as you might think. Because the program itself might somehow-- let's say, for example, have a log statement or function you could call to get the time that the program was launched or things like that.

But you're right. In practice, if you can use something like, let's say, the hardware system plot, which is often the lowest level, better system of timing of it-- yes, that kind of thing might work.

AUDIENCE: But even if you can pull the logs, it still depends on exactly what time you refuse a request. And so if you don't have control over how long it takes for your requests to get from your computer to the server, then I don't think you can deterministically guess exactly the right time.

PROFESSOR: That's right. That's right. The devil's in the details with all this kind of stuff.

In other words, if there's some way for you to figure out, for example, that type of timing channel, you might find out that the amount of entropy-- the amount of randomness-- is not, let's say, the full size of a timestamp, but maybe something that's much smaller. Because maybe the attacker can figure out the hour and the minute in which you did this, but not the second, for example. We'll take one more question, then we'll move on.

AUDIENCE: For the record, trying to roll your own randomness is usually a bad idea, right?

PROFESSOR: That's correct.

AUDIENCE: Usually, you should just use whatever's supplied by your systems.

PROFESSOR: Oh, yes. That's very true. It's like inventing your own cryptosystem, which is another popular thing undergrads sometimes want to do.

We're not the NSA, we're not mathematicians. That typically fails. So you're exactly right about that.

But even if you use system-supplied randomness, you still may end up with fewer bits of entropy than you expect. And I'll give you an example of that when we talk about address phase randomization. So that's basically how the stack canary approach works.

And so since we're in a security class, you might be wondering, so what kinds of things will stack canaries not catch? So when do canaries fail?

One way they can fail is if the attacker the things, like function pointers. Because if function pointers get [INAUDIBLE], there's nothing that the canary can do to prevent that type of exploit from taking place. For example, let's say you have code that declared a pointer.

It gets initialized in some way, it doesn't really matter. Then you have a buffer here. Once again, the gets function rears its ugly head.

And then, let's say, down here, we assign some value 5 for the pointer. Now note that we haven't actually tried to attack the return address of the function that contains this code. When we view the buffer overflow, this pointer address up here is going to get corrupted.

And so what ends up happening is that if the attacker can corrupt that pointer, then the attacker's able to write 5 to some attacker-controlled address. Does everyone see how the canary doesn't help here? Because we're basically not attacking the way that the function returns.

AUDIENCE: But won't the pointer be below the buffer?

PROFESSOR: So, yeah.

AUDIENCE: Not necessarily--

PROFESSOR: So you're worried about, is it going to be here, or is it going to be here?

AUDIENCE: I'm worried about when you-- will you actually be able to access where the pointer is when you're overturning--

PROFESSOR: Ah, yeah. So you can't necessarily-- that's a good question. So I think, in a lot of the previous examples, you've been assuming that this guy would be here.

Like, in the [INAUDIBLE]. If the stack is going this way, then the pointer would be down here. But the order of the particular variables-- it depends on a bunch of different things. It depends on the way that the compiler lays stuff out. It depends on the column dimension of the hardware, so on and so forth. But you're right that if the-- basically, if the buffer overflow was going this way, but the pointer was in front of the buffer, then it's going to work.

AUDIENCE: Why can't you associate a canary with the function canary, just like you did with the return address?

PROFESSOR: Ah. That's an interesting point. You could do those things. In fact, you could try to imagine a compiler that, whenever it had any pointer whatsoever, it would always try to add padding for various things. Right?

As it turns out, it seems like that will quickly get expensive, in terms of all the code that's added, to have to check for all those types of things. Because then you could imagine that every single time you want to invoke any pointer, or recall any function, you've got to have this code that's going to check whether that canary is correct.

But yeah, in principle, you could do something like that. So does this make sense? So we see that canaries don't help you on this equation.

And so another thing, as we've discussed before, is that if you can guess the randomness, then, basically, the random canaries don't work. Producing secure sources of randomness is actually a topic in and of itself. That's very, very complicated, so we're not going to go into great depth about that here. But suffice it to say, if you can guess the randomness, everything falls apart.

AUDIENCE: So do canaries usually have less bits than the return address? Because otherwise, couldn't you just memorize the return address and check that the address changed?

PROFESSOR: Let's see. So you're saying if the canary here is, let's say, smaller than--

AUDIENCE: I'm saying for the canary is that you know what that value is [INAUDIBLE]. Can't you also memorize the return address value and check if that's been changed?

PROFESSOR: Oh, so you're saying can't the secure system-- can't it look at the return address and figure out if that's been changed. Yeah. In other words, if there-- well, yes and no. Note that there's still this that's going get overwritten in the buffer overflow attack. So this may still cause problems.

But in principle, if somehow these things were invariant somehow, then you could do something like that. But the problem is that, in many cases, this return-- the bookkeeping overhead for that would be a little bit tricky. Because you can imagine that particular function may be called from places, and so on and so forth. Just in the interest of time, we're going to zoom forward a little bit. But if we have time at the end, we'll come back to some of these questions.

So those are some situations in which the canary can fail. There's some other places in which it can fail, too. For example, one way that it might fail is with malloc and free attacks. This is a uniquely C-style attack. Let's see what happens here.

Imagine that you have two pointers here, p and q. And then imagine that we issue a malloc for both of these. We give p 1,024 bytes of memory. We also give q 1,024 bytes of memory. And then, let's say that we do a strcpy on p from some bug that's controlled by the attacker.

So here's where the overflow happens. And then let's say that would be free q and then let's say that would be free p. OK. So it's fairly straightforward code, right? Two pointers-- malloc's the memory for each one of them. You use one of these on site functions, the buffer overflow happens, and we free q and we free p.

Let's assume that p and q-- the memory that's assigned to them-- are nearby, in terms of the layout in terms of [INAUDIBLE]. So both of these objects line next to each other in the memory space.

There's some subtle and evil things that can happen, right? Because this third copy here might actually over-- it'll fill p with a bunch of stuff, but it might also corrupt some of the state that belongs to q. OK? And this can cause problems.

And some of you may have done things in this unintentionally in your own code, when you have some type of weird use of pointers. And then stuff seems to work, but when you call free later on, it segfaults or something like that. Right? What I'm going to talk about here is the way that the attacker can take advantage of that behavior. We're actually going to explain why that happens.

So imagine that inside the implementation of free and malloc, an allocated block looks like this. So let's say that there is the app-visible data that lives up here. And then let's say you had a size variable down here. This is not something that the application sees directly. This is like some bookkeeping info that the free or the malloc systems attract so that you know the sizes of the buffer that it allocated.

Let's say that free block has some metadata that looks like this. You've got the size of the free block here. And then you've got a bunch of empty space here.

Then let's say-- this is where things get interesting. You've got a backwards pointer and then you've got a forward pointer. And maybe you've got some size data here. Now why are we having these two pointers here? It's because the memory allocation system, in this case, is using a doubly-linked list to track how the free blocks related to each other.

So when you allocate a free block, you take it off of this doubly-linked list. And then when you deallocate it, you do some pointer arithmetic, and then you fix these things up. Then you add it back to that linked list, right?

So as always, whenever you hear pointer arithmetic, you should think it's your canary. Because that's where a lot of these problems come about. And so the thing to note is that we had this buffer overflow here, the p. If we assume that p and q are next to each other, or very close in memory, then what can end up happening is that this buffer overflow can overwrite some of this size data for the allocated pointer, q.

Is everybody with me so far? Because if you're with me so far, then basically, you can use your imagination at this point and see where things go wrong. Because essentially, what's going to end up happening is that these free operations-- they look at this metadata to do all kinds of pointer manipulations with this kind of stuff.

Somewhere in the implementation of free, it's going to get some pointer based on the value of size, where size is something the attacker controls. Because the attacker did the buffer overflow. Right?

So then, you can imagine that it does a bunch of pointer arithmetic. So it's going to look at the back in the four pointers of this block. And then it's going to do something like update the back pointer. And also update the forward pointer.

And the exact specifics of this-- you don't need to worry about. This is just an example of the code that takes place in there. But the point is that note that because the attacker's overwritten size, the attacker now controls this pointer that's passed into the free code. And because of that, these two statements here, these are actually pointer updates. Right?

This is a pointer somewhere. And because the attacker has been able to control this p, he actually controls all this stuff, too. This is where the attack can actually take place.

So when the free code operates and it tries to do things like, for example, merge these two blocks, that's typically why you have [INAUDIBLE] doubly-linked list. Because if you have two blocks that are facing to each other and they're both free, you want to merge them to one big block. Well, we control size. That means we control this whole process here. That means if we've been clever in how these overflows are working, at these points, we can write to a memory in the way that we choose.

Does that make sense? And like I said, this type of thing often happens in your own code when you're not getting very clever with pointer. When you make some mistake with the double freeing or whatever, this is why stuff will segfault sometimes. Because you've messed up this metadata that lives with each one of these allocated blocks.

And then at some point, this calculation will point to some garbage value, and then you're dead. But if you're the attacker, you can actually choose that value and use it for your own advantage.

OK. So now let's get to another approach for getting rid of some of these buffer overflow attacks. And that approach is bounds checking.

The goal of bounds checking is to make sure that when you use a particular pointer, it only refers to something that is a memory object. And that pointer's in the valid bounds of that memory object. So that's the basic idea behind the idea. It's actually pretty simple-- at a high level.

Once again, in C, though, it's very difficult to actually understand things. Like, what does it actually mean for a pointer to be in bounds or out of bounds, or valid or invalid? So for example, let's say that you have two pieces of code like this. So you declare a character array of 1,024 bytes. And then let's say that you use something like this.

You declare a pointer, and then you'd get the address of one of the elements in x. Does this make sense? Is this a good idea to do that? It's hard to say.

If you're treating this x up here as a string, maybe it makes sense for Jim to take a pointer like this. Then you can increment and decrement, because maybe you're looking for some special value of your character in there.

But if this is a network message or something like that, maybe there's actually some struct that's embedded in here. So it doesn't actually make sense to walk this character by character, right? So the challenge here is that, once again, we can see it allows you to do whatever you want. It's hard to determine what it is you actually want it to do. And so, as a result, it's a little bit subtle with how you define things like pointer safety in C.

You can also imagine that life gets even more complicated if you use structs and unions. Imagine you had a union. It would look like this.

It's got some integer value in there. And then you've got some struct. And then, it has two integers inside of it.

Don't forget the way that the unions work is that, basically, the union's going to allocate the maximum size for the largest element. At any given moment, you typically expect that either this ni will be valid or this struct s will be valid, but not both. So imagine that you had code that did something like this. You get a pointer to address this guy. So I get an integer pointer to the address of, in the union, this struct, and then k.

Well, this reference is strictly speaking in bounds. There's memory that's been allocated for this. That's not incorrect. But are you actually, this moment in program of execution, treating this union as one of these guys or one of these guys? It's hard to say.

So as a result of these ambiguous pointers semantics that can arise in these C programs, typically, these bound checking approaches can only offer a weaker notion of pointer correctness. And so that notion is as follows.

If you have a pointer p prime that's derived from the base pointer p, then p prime should only be used to deference memory that belongs to the original base pointer. So for a derived pointer p prime that's derived from some original p, then p prime should only be used to deference memory that belongs to p. Know that this is a weaker goal than enforcing completely correct pointer semantics.

Because for example, you could still have weird issues like with this union here, for example. Maybe at this particular point in the program, it wasn't correct for the program to be able to reference that particular value in the union. But at least this pointer reference is imbalanced.

So maybe-- like this example up here-- maybe this creation of this pointer here violated the semantics of the network message embedded in x. But at least you're not trampling on arbitrary memory. You're only trampling on the memory that belongs to you. And so, in the world of C, this is considered success. So that's the basic idea.

Now, the challenge with enforcing these types of semantics here is that, in many cases, you need help from the compiler. So you need help from the compiler. You typically need to recompile programs to enforce these semantics. That can be a drag for backwards compatibility. But this is the basic notion of bounds checking.

What are some ways that you can implement bounds checking? One very simple way is this notion called electric fencing. The notion here is that, for every object that you allocate on the heap, you allocate a guard page that's immediately next to it.

And you set the page protection on that page, such that if anybody tries to touch that, you get a hard fault. The hard rules say that's out of bounds, and then the program will stop right there. And so this is a very simple thing that you can do. And what's nice about this approach actually, is that whenever you have an invalid memory reference, this causes a fault immediately, right. If you've ever debugged the Base C or C++ program, one of the big problems is that a lot of times when you corrupt memory, that memory is corrupted silently, and for a while, and it isn't until later that something crashes and then only then you realize something happened. But you don't know what that something is. You simply do what they call heisenbugs, right. Things that have this notion of uncertainty in them. So what's nice about this is that as soon as the pointer hits here, boom, it's a guard page, everything blows up.

Now can you think of a disadvantage with this approach?

AUDIENCE: It takes longer [INAUDIBLE].

PROFESSOR: Yeah exactly. So imagine that this little-- this key thing here was super, super small, then I've allocated a whole page just to make sure that my little tiny thing here didn't get-- didn't have one of these pointer attacks. So this is very space intensive. And so-- but people don't really deploy something like this in production. This could be useful for the bugging thing, but you would never do this for a real program. So that make sense? So these electrical fences are actually pretty-- pretty simple to understand.

AUDIENCE: Why does that have to be so large, necessarily?

PROFESSOR: Ah, so the reason is because this guard page here, you're typically relying on the hardware, like page level protections to deal with those types of things. And so there's like certain memory size you can set to the size of the page, according to [? Hollis ?]. But typically that page is 4k, for example. So getting back to your question, this is some like super small value here, then yeah [INAUDIBLE] 2 bytes where you got 4k here protecting it.

AUDIENCE: In protecting [INAUDIBLE] individual [INAUDIBLE].

PROFESSOR: Oh sorry yeah, yeah so by heap I mean like heap object.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah thank you for-- yeah exactly. So imagine like for each malloc you do, you can have one of these-- and set the guard page for it.

AUDIENCE: And you do it for log and above? Or just above?

PROFESSOR: You can do either.

AUDIENCE: [INAUDIBLE]

PROFESSOR: That's right.

AUDIENCE: [INAUDIBLE].

PROFESSOR: That's right, well you could do either. The ones we have depending on this-- on the size of the object. I mean now you got to declare two guard fences, right. So now this quickly gets out of control. Which yeah, you could have a booking [INAUDIBLE]. So that's the basic idea behind that.

And then another approach you can look at is what they call fat pointers. And so the idea here is we actually want to modify the pointer representation itself to include bounds information in it. So if you look at your regular 32-bit pointer what's it look like? Well the answer is, 32-bits. And then you got [INAUDIBLE]. Right? If you look at a fat pointer then one way you can think about looking at this is you got a 4 byte base. And then you have a 4 byte end.

So in other words, this is where it would allocate out that it starts, that's where it ends and then you've got a 4 byte cur address. So this is where the pointer actually is, within that bounds, right. So basically what happens is that the compiler will generate code, such that when you access these fat pointers this gets updated, but then it'll also check these two things to make sure that nothing bad has happened during that upgrade.

So for example you can imagine that if I had code like this. So I have an end pointer and then I allocate 8 bytes. So assuming that we're on a 32-bit architecture to point to 2 [INAUDIBLE]. And then I have some while loop that it is going to just assign some value to the pointer and then increment the pointer-- what you'll see is that the current address for this pointer, like at this point in code, will point to the base, right. And then every time we iterate through here, we can see that we're either checking a bound, or incrementing a bound.

So at this point we want to dereference it. We can actually check and see, is the current address at that pointer, in this ring. And if it's not you throw in an exception here and so on and so forth. So once again, where is this taking place? This Is taking place in new code that the compiler generated.

So one question that came up on the online discussion group, some people were saying, well what if it's instrumented code, what does that mean, right? So when I say that the-- that the compiler generates new code, imagine that there-- this is what you see as a programmer. But before this operation actually takes place, imagine the compiler inserted some new C code here that basically looks at these base bounds here. And then if there was something out of bounds it would then do an exit, or an abort, or something like that. So that's what it means to say that there is instrumented code. It's that you take the source code, use the program of C, add some new C source code and then compile that video program. So the basic idea I think behind the fat pointer is pretty simple.

There's some disadvantages to this. The biggest disadvantage is that, oh my goodness look how big the pointers are now, right. And so what this means is that you can't just take a fat pointer and pass it to an unmodified, off the shell library. Because it may have certain expectations that pointers are a certain size and we give you this thing, it's just going to-- it's going to blow up. We also have trouble if you want to include these types of pointers and structs, or things like that. Because that can actually change the size of the struct, right.

So a very popular thing in C code to do is to take like the size of the struct and then like do something as a result of that. Like reserve some disc space for a struct of that size, and so on and so forth. So this causes all that stuff to blow up, right. Because once again, the pointers have gotten very, very big.

And another thing which is a bit subtle, but it's that these fat pointers typically will not be able to be updated in an atomic fashion, right. So on 32-bit architectures typically, if you do like a write to a 32-bit variable, that write is atomic, right. But now, these pointers are these three integer sized things, right. So if you have any code that takes advantage of the fact that it expects pointer writes to be atomic, then you may get in trouble, right. Because you can imagine that to do some of these checks, you have to look at the current address and then look at this and then you might have to increment that, and so on and so forth.

So this can cause very subtle concurrency bugs if you have code that depends on that atomacy of fail [INAUDIBLE]. So does that all make sense? So that's one approach you can do. But kind of like electric fences, this has some nasty side effects that means the people don't typically use that in practice.

So now we can start talking about bounds checking, with respect to the shadow of the infrastructure that I mentioned in the baggy bounds paper. So the basic idea for the shadow base structure is for each object that you allocate, you want to store how big the object is. Right, so for example, if you have some pointer that you call malloc on right, you need to store that size of that object there, and then note that if you have some thing that's like a static variable like this, right, the compiler can automatically figure out what the bounds are for that thing there, statically speaking.

So for each one of these pointers you need to interpose somehow on two operations. Basically you do arithmetic. So this is things like q equals p plus 7, or whatever. And then you want to interpose on dereferencing. So this is something like q equals a or something like that.

So what's interesting is that you might think, well why can't we just rely on the reference when interposing stuff? Why do we have to look at this point arithmetic here? But similarly you might wonder the other thing. Like why can't you just deal with one of these non [INAUDIBLE] interpose [INAUDIBLE]? So you can't just signal an error if you see the arithmetic going out of bounds because in c that may or may not be there. So in other words, a very common medium is C and C++ is you might have a pointer that points to one pass the valid end of an object right, and then you use that as a stop condition, right. So you iterate to the object and once you hit that end pointer, that's when you actually stop the loop or whatever.

So if we just interpose on arithmetic and we always cause a hard fault, when we see a pointer go out of bounds, that may actually break a lot of legitimate applications, right. So we can't just interpose on that. And so you might say, well why can't you just interpose on the reference thing, and you just-- when we notice that you've cut something out of bounds, we'll just read there and there. Well the challenge there is that how do you know it's out of bounds? Right, it's the-- it's the arithmetic in our positioning that officially allows us to tell whether or not this thing's going to be legal here, right. Because it's the interpositioning on the arithmetic that allows us to track where the pointer is with respect to it's original baseline. So that's the basic idea there.

And so the next question is how do we actually implement the bounds checking? Because basically we need some way to map a particular pointer address to some type of bounds information for that pointer. And so a lot of your previous solutions use things like, for example, like a hash table, or a tree, right that will allow you to do lookups right, and stay the gray. So given a pointer address, I do some lookup in this data structure, figure out what the bounds are. Given those bounds I can then figure out if I want to allow the action to take place or not.

Now the problem with that is that it's a slow lookup, right because these data structures you're thinking it's a tree, or you're going through a bunch of branches before you can actually hit the value potentially. And even if it's a hash table where there's an overflow in the bucket you got to follow chains, or do you're code, or things like that. So the baggy bounds paper that we are about to look at actually figured out a very efficient data structure that tracked to these bounds, to make that bound checking very fat. So let's just step into that right now.

But before we go into that let me very briefly talk about how buddy allocation works. Because that's one of the things that came up in a lot of the questions. So one thing you will see for these papers is that a lot of times they are not self-contained, right. So they will mention things that they will assume that you know, but you may not know them. Don't get discouraged by that. That happens to me too sometimes. These papers are written in a way they assume a lot of prior knowledge, so don't get discouraged by that. Luckily we actually access to the internet we can look up some of that stuff. Can you imagine what happened in our parents time? They just didn't understand stuff they just had to go home, right. So don't be afraid to look stuff up to get to Wikipedia it's mostly correct.

So how does-- how does the buddy allocation system work? So basically what it does at first it treats unallocated memory as one big block. OK. And then when you request a smaller block for dynamic allocation, it tries to split that address base using powers of 2 until it finds a block that is just big enough to work. So let's say a request came in and say A is going to equal to malloc 28. 28 bytes. And let's just say this toy example is only 128 bytes of memory total. So the buddy allocator is going to look at this and say, well I have 128 bytes of memory, but it's too wasteful to allocate this whole thing to this 28 byte request.

So I'm going to split this request in two and then see if I have smaller block that's just big enough. So it's going to say, OK put this to 0 to 64 and 64 to 128. Ah OK, but this block here is still too big, right. Basically what the buddy algorithm wants to do is find a block such that the allocated data in the real object, 28 bytes, is at least half the size of that block. So buddy allocator says, OK this thing over here is still too big. So what it's going to do is it's going to split the memory space again, right. So from 0 to 32 and then it's going to say, ah OK 28 bytes that is more than half the size of this block here. So now this block is going to be allocated to A. OK, and so it gets this address here.

Now let's say that we have another question comes in for B and let's say we want to malloc 50 right. So what's going to happen is that the buddy allocator will say, ah OK I actually have a block here that's big enough, right. 50 Is greater than half the size of this thing so I'll just allocate that right there. So we have this system, or setup, where we have A here, and then we have B here, and then let's say we had another request that came in for 20 bytes. This is actually pretty straightforward because we can put that right here, right. So then you have something that looks like this.

Then what's interesting is that when you deallocate memory, if you have to deallocate a block that are next to each other and are the same size, the buddy allocator will merge them into a block that's twice as big, right. So if we had free let's say C then we go to this situation, we can't do any merging, because this is the only possible block that this one could have been merged with. It's the same size, but this things still occupied.

So then if we do a free on A, then we have this situation here. Right, where these two 32 byte blocks were merged into one size 64, and that this one, a size 64 is still out there. Right, so it's called the buddy system because once again, whenever you have two adjacent blocks that are of the same size and that could be merged to form an aligned block, then the system will merge that buddy with this other buddy and then create that new block that's twice as big. So the thing that's nice about this system is that it's very simple to figure out where buddy's are. Because you can do very cutesy arithmetic, like the buddy bounds system-- baggy bounds system works. But anyway I'm not going into details. This is basically how buddy allocation works. Does that make sense?

Right, and one question that came up a lot in all my discussions, isn't this wasteful? Right, so for example, imagine that up here at the beginning I had a request for size 65 bytes, right. So if I have a request for 65 bytes, I would allocate this whole structure up here and then there's-- actually you're out of dynamic memory and can't do any more allocations. And the answer is yes, that is wasteful. But once again, it's a trade off, right. Because it's very easy to do these calculations on how to do merging and stuff like that. So if you want finer grain allocation, there are other valid ones for that. It's outside the scope of the lecture so, we can buffer that offline if you want. That's basically how the buddy-- sorry the, the buddy allocator works.

So what is the baggy bounds system going to do? Well, it is going through a y, on couple of tricks. So the first idea is you round up each allocation to a power of 2, and you align the request to that power of 2. Right, so essentially the buddy allocators very nice because it handles a lot of that for you, right. It naturally will do that kind of thing. Because that's just the way that it allocates and deallocates to memory. And so the second thing that's going to happen, baggy bounds system, is you express each bound as log base 2 of the allocation size.

Right, and so what this means-- and so why can we do this? Well once again all of our allocation sizes are powers of 2, right. So we don't need very many bits to represent how big a particular allocation size is. So for example, if your allocation size is 16, then you just need four-- the log rhythm of that, 4 bits of the allocation size, right. Does that make sense?

Right, this another popular question here. This is why you only need small number of bits here, because we're basically forcing the allocation sizes hit this quantized way that you grow. Like if you could only have something, let's say 16 bytes or 32 bytes. You can't have for example, 33 bytes.

And then the third thing that baggy bounds is going to do is store the limit info in a linear array 1 byte per entry but we're going to allocate memory at the granularity of a slot. Which in the paper they used 16 bytes as the slot width. So for example, now this next one, this is 1 bit that wasn't actually specifically said in the paper which if you don't grasp it'll make the paper very tricky to understand, right. So now you can have a slot size which is equal to 16, so if you do p equals malloc 16 so what's going to happen? So in this bounds table you're going to say take that pointer plot it by plot size it's going to equal 4, right. So in that bounds table we're going to put the logarithm of the allocation size in the table. Does that make sense?

OK, now what the tricky thing is, let's say that you have something like this. Right, so let's say that you out 32 bytes. What is the bounds table going to look like there? So here we actually have to update the bounds table to abbreviate your p, or sorry t for the size you need. But that fit the bounds table twice. Right, once for the first slot memory that this allocation takes up. And then a second time for that second slot that it takes up. Right, so once again 32 is the allocation size. This is the log of that allocation size. So for the two slots that this memory takes up, we're going to update the bounds table twice. Does that makes sense? Right, and this is really the key that I think for a lot of people that's going to make the paper make sense or not make sense, right. Because that bounds table multiple times if any outside the allocation.

AUDIENCE: Can you repeat that for me again?

PROFESSOR: Excuse me?

AUDIENCE: Can you repeat that again?

PROFESSOR: Oh yeah, yeah, sure, sure. So basically what the idea is that I mean you've got this bounds table here and it's got a bunch of entries. But it basically needs entries to cover all of p size, all the allocation size. OK, so in this case it was very simple because basically this is just one slot, due to the size. Here it's multiple slot sizes, right. So what's going to happen is that imagine then that we had a pointer that's moving in the range of p. You have to have some of the back end table slot for each one of those places where p [INAUDIBLE], right. And so it's this second piece that makes the paper a little bit confusing I think. But it doesn't really go into depth about that, but this is how that works.

OK so armed with the bounds table stuff what happens if we have a C code that looks like this? So you have a pointer, p-prime, you derive it from p, we would add some variable i. So how do you get the size of the allocation belonging to p? Well you look in the table using this lookup here.

Right, so the size of the data that's been allocated to p is going to be equal to 1 and then when you Left Shift that by looking at the table, taking that pointer value, and then Right Shifting that by the log of the table size. Right, if the arithmetic works out because of the way that we're binding pointers to the table bounds, right. So this will get us-- this thing right here, will get us the log of the sides. And then this thing over here basically expands that into like the regular value, right.

So for example, if the size of this pointer were 32, in terms of bytes we've allocated, right. This is going to get us five when we look at the table, then when we Left Shift it this way, Left Shift the one this way, then we're going to get 32 back again from here. OK.

And then we want to find the base of that pointer. Take a pointer itself and then we're going to and that with the side minus 1. Now what this is going to do is, this is actually going to give us a mass, that you can think of it. And that mass is going to allow us to recover the base here. So imagine that your size equals 16. So 16 equals this in binary. Right, there's a bunch of zeros off this way. So we've got a 1 here, we've got some zeros over here.

So if we look at the bit-wide inverse of 16 minus 1, then-- actually sorry. So if we look at 16 minus 1, so what's that going to look like? 60 minus 1 we're going to look like right, something like this. OK. And if we take the inverse of that what is that going to give us? Right, in binary. So basically this thing here allows us to basically clear the bit that essentially would be offset from that valid pointer and just give us the base of that pointer. OK. And so once we've got this, then it's very simple to check whether this pointer's in bounds, right.

So we can basically just check whether p-prime is greater than or equal to base and whether p-prime minus the base is less than size. This is just a straightforward thing you do, right. Just seeing whether that derived pointer exists within the bounds of this [INAUDIBLE]. Right, so at this point things are pretty straightforward. Now they have like a optimized check in the paper, I'm not going to go into that detail. But suffice it to say that all the binary arithmetic, it resolves down to the same thing. There's just some clever tricks there to avoid some of the explicit calculations we do here. That's the basic idea.

And so the fifth trick that the baggy bounds system uses is that it uses the virtual memory system to prevent out of bounds [INAUDIBLE] right. So the idea here is that-- how much time do we have by the way? Probably like zero? So the basic idea here is that if we have a pointer [INAUDIBLE] here, that we detect is out of bounds, what we can do is actually set the high order bit on a pointer, right. And by doing that we guarantee that pointer is dereferenced, then the caging hardware's going to be [INAUDIBLE], we're going to throw a hard error, right. Now in and of itself, just setting that bit does not cause a problem. It's only when you dereference that pointer that you get into problems. OK?