The following content is provided under a Creative Commons license. Your support will help MIT OpenCourseWare continue to offer high quality educational resources for free. To make a donation, or to view additional materials from hundreds of MIT courses, visit MIT OpenCourseWare at ocw.mit.edu.

PROFESSOR: All right, let's get started. So welcome to the next lecture about exploiting buffer overflow. So today, what we're going to do is, we're going to finish up our discussion about baggy bounds and then we're going to move on to a couple of other different techniques for protecting its buffer overflows.

Then we're going to talk about the paper for today, which is the blind return oriented programming. So if you were like me when you first read that paper you kind of felt like you were watching like a Christopher Nolan movie at the beginning. It was kind of like mind blowing right.

So what we're going to do is we're going to actually step through how some of these gadgets work right. And so hopefully by the end, you'll be able to understand all this sort of high tech chicanery that they're doing in the paper.

So first of all, like I said, let's just close up with the baggy bounds discussions. Let's go through a very simple example here. So let's say that we're going to define a pointer called P. And let's say that we're going to give it allocation size of 44. Let's also assume that the slot size equals 16 bytes OK.

So what's going to happen when we do this malloc up here? So as you know, the baggy bounds system is going to pad that allocation out to the next power of two, right. So even though we've only allocated 44 bytes here, we're actually going to allocate 64 bytes for this pointer up here. And so also note too, this is the slot size is 16.

How many bounds tables entries are we going to create? Well we're going to create the allocation size, which in this case 64, divided by the slot size, which is 16. So in this case we'll create four different bounds table entries for this thing right here.

Each one of those entries is going to be set to the log of the allocation size, which in this case is going to be 6 right. Because the allocation size is 64, OK? So, so far so good.

Then we're going to define another pointer called q and we're going to set it equal to p plus 60. So what happens when we do this? Well note that strictly speaking, this access is out of bounds, right. Because this was only allocated 44 bytes of memory, but of course the way that baggy bounds works is that it will actually allow axises that are out of bounds, if they stay within that baggy bounds. So even though strictly speaking, the programmer probably shouldn't have done this, this is actually going to be OK, right. We're not going to raise any flags or anything like that.

Now let's say that the next thing we do is we need to find another pointer, which is going to be set equal to q plus 16 right. Now this is actually going to cause an error, right. Because now q is at an offset of 60 plus 16, which equals 76. So this is actually 12 bytes away from the end of that baggy bounds. OK? And that's actually greater than half a slot away.

All right, so if you remember the baggy bounds system will actually throw a hard synchronous error if you get beyond 1/2 a slot from the edge of that baggy bounds. So this will actually cause the program to fail. This will actually make it stop.

Now let's imagine that we didn't have this line of code in the program, OK. So we had these two, but we don't have this one. So what if we instead of doing this line, did something that looks like this. We declare another pointer, let's call it s, and we set it equal to q plus 8.

Now in this case, the pointer is going to be at 60 plus 8, which equals 68 bytes away from p, right. So this is only four bytes beyond that baggy bound. So this will not actually cause an error. Even though it is strictly speaking, out of bounds. What we will do here though, is set that high order bit on the pointer, right. So that if anyone subsequently tries to dereference this thing, it's going to cause a hard fault at that point.

And then let's say, the final thing that we do is we declare another pointer t, which is going to equal s minus 32. So what happens here is that essentially we brought this pointer t, it is now back in bounds, right. So what that means is that even though this guy was out of bounds, now we sort of going back to the original allocated region, that we originally created up here.

So as a result, t will not have that high order bit step and so you can dereference T and everything will be fine. So does this all make sense? This should be fairly straightforward.

AUDIENCE: [INAUDIBLE] the difference between r and s, how would you know that r is-- or how does the program know that r is 1/2 the [INAUDIBLE].

PROFESSOR: So note that, like up here, when we create r you can basically interpose, we get an instrumented code that's going to be working at all of these pointer operations. So basically we can tell is that we know where P is going to be. I'm sorry, we know where q is going to be. And we know that q is within those baggy bounds.

And so when we do this operation here, the instrumentation of baggy bounds adds and we're able to say, aha, well I know where that source formula is coming from. And then if you look at this offset here, you determine it's more than a 1/2 slot away from slot side.

So basically what you think about is that as we're doing these pointer operations, and looking and saying is how are you going out of bounds, have you gone out of bounds, yes or no. At some point you're going to have some operation that's going to involve a pointer that is either in bounds within the baggy bounds and then some thing over here that makes it go out of bounds. So at that moment, right when that happens, that's how we know that something chicanerous has arisen.

All right so, hopefully that should all make sense. And so this is very briefly a review of the homework question. So hopefully you can understand this and our homework question should be pretty easy to understand. So we have a character pointer the malloc had 256 bytes to it, Then we declare a character pointer q, that is equal to

that pointer plus 256 and then we essentially try to dereference this pointer.

So what's going to happen? Well note that this is an exact power too, right. So there's not actually any bagginess in the bounds, right. So when we do this right here, this makes q point to one pass the end of those baggy bounds. So just like in this example up here, this line is actually fine, but it will cause the high bit to be set in q, right. So when you come down here and reference it, then everything blows up and it's time to call in your insurance agent. So pretty straightforward?

OK so, that's basically two examples that you can flavor for how baggy bounds works. As I mentioned in the last lecture, you don't actually have to instrument every pointer operation, if you can use static code analysis to figure out the particular set of pointer operations is safe. I'll defer further discussion of some of the static analysis [INAUDIBLE], but suffice it to say that you don't always have to have all this bit wise arithmetic that you have in some of the cases that we've examined before.

And so another question that came up a lot in Piazza was, how does baggy bounds ensure compatibility with these preexisting, non-instrumented libraries, right. And so the Piazza idea behind how baggy bounds does that is, that when baggy bounds initializes the bounds tables, they set all the entries to be that bound of 31. So when we read the bounds table, each entry represents 2 to the power of that entry, the size of that particular pointer.

So by initializing all those bounds of 31, what this allows us to do is automatically assume that each pointer from [INAUDIBLE] the code is going to have the largest bound possible, 2 raised to the 31. So let me just give you a very simple example here that will hopefully make this a little clearer.

So let's say that this over here is the memory space that we lose for heap. This is simple example, let's suppose that basically what this memory space [INAUDIBLE] two components. This is the heap, that is out by the unistrumented code. And then let's suppose that down here we have the heat that is allocated by the instrumented code.

So what's baggy bounds going to do? So remember, baggy bounds has this notion of a slot size, right. So basically the slot size is 16, you only have entry for every sort of slot of size 16 over here. So basically the bounds table in this case, you can think of being set up into three places, sorry two places. So initially all of the bounds table, all the entries are initialized to 2 to the 30-- or sorry, to the 31.

But then eventually as the instrument code runs it's actually going to use the baggy bounds algorithm to set these values for whatever should be appropriate for that particular [INAUDIBLE], right. So what ends happening is that if you did-- if instrumented code gets a pointer that comes from here, then those baggy bounds with each particular pointer will always be set to the largest possible value, 231. 2 to the 31, right. Which means that it's going to be impossible for baggy bounds, entry of the code, to think that you've done an out of bound operation with that pointer that's coming from this uninstrumented library.

So does that make sense? So the idea is that in instrumented code we're always going to be doing these comparisons with the pointers, but if we always set the bounds entries for uninstrumented pointer code 2 to the 31, you can never have a dereference error.

OK so that's basically how we have this nice interoperability between the entry of the baggy bounds code in between a noninstrumented off the shelf legacy library's. So putting it all together, what does this all mean? So, we have this system here which is nice because it doesn't make the uninstrumented libraries blow up, but one problem is that we can't detect out of bounds pointers that were generated in the uninstrumented code, right. Because we're never going to set that high bit for example, if that [INAUDIBLE] pointer gets too big, or gets too small or anything like that. So we actually can't provide memory safety for operations that take place in uninstrumented code.

You also can't detect when we pass an out of bounds pointer from instrumented code to uninstrumented code. Something insane could happen, right. Because remember if you had this out of bounds pulled it from the instrumented code it has

that high bit set to 1, right. So it looks like it's super ginormous.

Now we know if we just kept that code in instrumented code, we might clear that flag at some point if it comes back in bounds. But if we just pass this ginormous address to uninstrumented code, then who knows, it may try to dereference it, it may do something crazy.

It may even bring that pointer back in bounds, but we would never have an opportunity to clear that high bit, right. So you can come up-- you still may come up with some inter-op issues there, even if we use this scheme over here.

OK, so that's essentially how baggy bounds works on a 32-- you got a question?

AUDIENCE: Yeah, so if you have a instrumented coding meets like allocated memory, is it using the same malloc that the attributing code is using, or?

PROFESSOR: Yeah so it's a bit subtle. So like in this case here, it's like very stark what's going on, because there's just two regions, one of which is used by each set of things. So it actually depends on the if they use [INAUDIBLE] and stuff like that. You can also imagine that like in C++ [INAUDIBLE] for example, you can define your own allocator, right. So it kind of depends [INAUDIBLE].

AUDIENCE: [INAUDIBLE] input the same, how does the allocator know whether or not to set 31 or [INAUDIBLE].

PROFESSOR: Yeah so at the lower level, typically the way that these allocation algorithms work, is that you call unknown system [INAUDIBLE] or something like that, sort of move a pointer up. So you can imagine if you have multiple allocators, all trying to allocate memory, they each have their own chunk of memory they reserve for themselves basically, right. So in real life it may be more fragmented than this, that's essentially on a high level, how it works.

OK so this was a baggy bounds on a 32-bit system. So as you all know 64-bit systems are the bees knees these days, so how does baggy bounds work on those systems? Well, in those systems you can actually get rid of the bounds table,

because we can actually store some information about the bounds, from the pointer itself.

So imagine we're going to look at a regular pointer in a baggy bounds system. So we can use it, like this. So we can-- if the pointer's in bounds, we can basically just set the first 21 bits to 0. We can put the size in these 5 bits here. And once again this is representing the log base 2 at the size here. And then we have here, in the remaining 38 bits, just the regular address bits.

Now the reason why this doesn't massively curtail the address size of the program you use, is that a lot of these high order bits, the operating system and-or the hardware, doesn't let a application use, for various reasons, right. So as it turns out, we're not dramatically shrinking the amount of [INAUDIBLE] application you use in the system. This is what a regular pointer looks like.

Now what happens when we only have one of these out of bounds pointers? Well, in a 32-bit system all we can do basically is just set that high order bit and you just hope that thing never got beyond a 1/2 a slot away from it's base. But now that we have all this extra address space here, you can actually put the out of bounds offset directly in this pointer. So we can do something like this.

So we can have 13 bits here for the offset, right, the out of bound offset. How far away is this out of bounds pointer, from the place where it should be? And then once again you can put the actual size of the referred object here. This will be 0 once again. And this will be the real address base here. And so this may be reminiscent to you of some type of fact pointer representation, but there's a couple of advantages here, now that we're moving in the 64-bit world.

So first of all, you'll note that these tag pointers, these are the regular size of a regular pointer, right. Pointer's are still just 64-bits wide in both of these setups. So that's nice because that means for example, that means and rights to that pointer are time. Unlike in traditional fat finger world, where you actually have to use multiple words represent that fat pointer. So that's nice.

And also note that we can trivially ask these things, uninstrumented code, because they work and are the same size as regular pointers. We can put these things in structs for example, and the size of those structs won't change. So this is very nice if we can work in that 64-bit world. So does that all make sense?

AUDIENCE: So why are there eight 0-bits [INAUDIBLE] pointer there? Where like the 5 size bits previously weren't.

PROFESSOR: So you're talking about down here?

AUDIENCE: Yeah, is there a reason why we can't just store a [INAUDIBLE] if we're like six 0-bits there and had more bits for the offset, like why is the number 8?

PROFESSOR: So I think so in some cases there are certain line issues that we have to work with. The [INAUDIBLE] issue is to deal with if the bits are higher. I don't think, in principle, there's any reason why you couldn't read some of these things [INAUDIBLE]. Well there may be some hard versions that I'm not thinking of right now, but [INAUDIBLE] some of these would have to 0 or otherwise the hardware's going to cause a problem. Any other questions?

OK so, next thing are you wondering is, can you still launch a buffer overflows in the baggy bounds system, obviously because I gave you another paper to read, so clearly this thing, this doesn't solve all the problems, right? So one problem you might run into is that if you have uninstrumented code once again, we can't detect any problems in uninstrumented code.

You could also encounter memory vulnerabilities that come about from the dynamic memory allocation system. So if you can remember in the lecture we looked at this weird free malloc weird pointer thing that took place. Baggy bounds won't necessarily prevent you from some of that stuff.

We also discussed last lecture, where the fact that code pointers do not have bounds associated with them, right. So now you have struct that has a buffer at the bottom, it has a function pointer up top, if you have a buffer overflow in to that function pointer, right.

Let's say that buffer overflow is still within the baggy bounds. So you've overridden that function pointer. We would try to execute that function pointer, it could be pointed at something for, attack a control piece of memory. OK, and bounds won't help with that, because there's no bounds associate with function pointers.

And so in general, what are the cost's of baggy bounds? So there are essentially four. So the first cost is space, right. So if you're using a fat pointer, obviously you've got to make pointers bigger. But if you're using the baggy bounds system that we just discussed, you've got to store the bounds table, right. And so the bounds table has that slot size which allows you to control how big that bounds table is, but still you may end up using [INAUDIBLE] memory for that.

You've also got the CPU overhead of doing all of the pointer instrumentation, right. So for every, or close to every pointer thing that you do, you got to check these bounds using those shift operations and things like that. So that's going to slow your program down.

There's also this problem with false alarms, right. So as we discussed, it may be the case that a program generates out of bound pointers but never tries to dereference it, right. Strictly speaking that's not an issue. The baggy bounds will flag the creation of those out of bounds pointers, if they get beyond a 1/2 a slot size, at least in the 32-bit solution, right.

And so what you'll see with a lot of security tools, is that false alarms really reduce the likelihood that people are going to use your tools, right. Because in practice we would all hope that we care about security, but actually what do people care about?

They want to be able to upload their silly Facebook photos and life things, and they want to be able to make things go fast and stuff like that. So you really want your security tools to probably have less coverage of finding bugs, but actually have 0 false alarms. As opposed to catching all types of security vulnerabilities, but then maybe having some false alarms that are going to irritate developers, or irritate users.

And the other costs that you have for this is finally, is that you need compiler support, right. Which can actually end up being nontrivial, because you have to go in there, you have to add all the instrumentation, crawl the pointer checks, and so on ans so forth. So those are basically the cost of these bounds checking approaches. So that concludes the discussion of baggy bounds.

And so now we can actually think about a two other mitigation strategies for buffer overflows. They're actually much simpler to explain and understand.

So one of these approaches is called a non-executable memory. And the basic idea is that the paging hardware is going to specify 3-bits for each page that you have in memory, read, write and execute, right. Can the program read that memory, write to it, execute it. The first 2-bits are old, they've been around for a while, that last bit is actually a fairly new construction.

And so the idea is that you can actually make the stack non-executable, right. So if you make the stack non-executable that means that the adversary can't run code just by pointing-- by creating that shell code and then sort of jumping to someplace in that buffer.

And so what a lot of systems do, is they actually specify a policy like this. So right exclusive or x, which means that if you have a particular page, you can either write to it, or you can treat it as executable code, but you cannot do both. OK and so that once again, is going to prevent the attacker from just putting executable code in the stack and then going straight to it. So this is-- should be pretty straightforward, right. So we've removed, at the hardware level, this attack vector of the attacker putting executable code in the stack. So what's nice about this?

Well potentially this works without any changes to the application, right. This is all taken place at the hardware level and at the OS level, with the OS just making sure the pages are protected with these bits, OK. So that's very, very nuts, right. Because you don;t have to worry about this compiler support issue we had over here.

The other nice thing is that, as I mentioned in the last lecture, the hardware's always watching you, even though the OS is not, right. So these bits being said over here, you know they're looked at and verified for correctness at every memory reference that you make by the code. That's a very nice aspect of this too.

Now one disadvantage of this system though, is that it makes it harder for an application to dynamically generate code, in benign or benevolent cases. And the best example of that is, the just-in-time compilers that we discussed from last lecture, right.

So how is it that you can go to a web page and your JavaScript code executes fast. It downloads that JavaScript source, it initially probably starts just interpreting it, but then at some point it's going to find some hot path, some hot loop and then it's going to dynamically generate x86 machine code and execute that directly, right. But to get that to work you have to be able to dynamically write code to a page.

So there's some ways you can get around this for example, you could imagine that the just-in-time compiler initially sets the write bit and then it removes the write bit, then it sets the execute bits. There are some ways that you can get around that, but it can be a little bit tricky sometimes. On a higher level, that's how non-executable memory works, pretty easy to understand.

AUDIENCE: What is the definition of like executable instructions? So if you change the attenuator [INAUDIBLE] it's not considered a executable instruction.

PROFESSOR: Well basically no. Can you set like the instruction pointer register to that value. In other words, can you-- if you have a bunch of memory pages, can you actually set EIP there and actually start executing code from that page.

AUDIENCE: Ah.

PROFESSOR: OK, so that is nonexecutable memory.

And so another technique you might imagine for protecting against a buffer overflows is using a randomized addresses or address spaces. And so the

observation here is, that a lot of the attacks that we've discussed so far use hard coded addresses, right.

And so if you think about a lot of the attacks you've been working on in your lab, how does that work? You open up the program in GDB, you find out the location of some things, you may create some shell code that actually has some hard coded addresses in there, right.

So the idea behind the randomized address base is simple. Basically you want to make it difficult for the attacker to guess addresses. So there's a couple different ways you could think about doing this, right. So one idea is that you can imagine having stack randomization, right.

So imagine that from here to here, this is the entire virtual memory space of the program, right. As we described this stuff to you so far, basically the stack always starts with this particular place up here, always goes down, right, and the program codes down here and the heap always goes up here. And all these seg-- all of these segments, the stack, the heap, and the program code, they all start at a well known location.

So imagine for example, like if my lecture notes here are the stack. You can imagine instead of the stack always starting here at this known location, maybe you start it here, maybe you start it here. Somewhere else like that, right. Similarly you can imagine that maybe the program code which used to always start down here, maybe we start it up here, or down here, or something like that, right.

So the idea now is that if you, the attacker, control one of these binary's, you can look in GDB and figure out where all these offsets are, but they're not actually going to help you figure out where those offsets are in the real code that's running on the server, right. So that's the basic idea behind these randomized address spaces there.

And so this takes advantage of the fact that a lot of the code that you generate, doesn't have to be loaded into a specific place in memory, right. So unless you're

writing like a device driver, or something like, that maybe is interacting with some hardware that requires this particular address to be belong in this particular buffer so it can copy information in. If you're not doing stuff like that then typically your codes going to be relocatable. So this approach will work very nicely with that kind of stuff.

So once again the question is, can you exploit this? Obviously the answer is still yes. There's a couple different ways you can do it as we'll discuss later today in the [INAUDIBLE] paper, the attacker can actually extract randomness, right. And so in general that's how you defeat all these randomized approaches. You make them unrandom, by either finding out the random seed that the attacker was doing, or by somehow leveraging the fact that the attacker leaks information about the randomized locations of these things.

And another thing that's interesting is that for a lot of the attacks we've discussed so far, we've basically been using these sort of hard coded addresses, but note that the attacker may not necessarily care about jumping to a specific address. Or there's this attack called a heap attack, which is actually pretty hilarious if you're a bad person, I suppose.

So the way that this heap attack works is, that the attacker essentially just starts dynamically allocating a ton of shell code and just stuffs it randomly in memory, right. This is particularly effective if you're using like a dynamically high level language like JavaScript let's say.

So the tag reader is sitting in a tight loop and just generate a bunch of shell code strings, right. And you just fill the heap with all these shell code strings, right. Now the attacker maybe cannot figure out where the exact location is of each of the shell code strings, but if you've allocated 10s of megabytes of shell code strings and then just do a random jump, right.

If you could somehow control one of these ret pointers, then hey, maybe you'll land in shell code, right. And one trick you can actually use is this thing called NOP sleds, which is also pretty hilarious. So imagine that if you have a shell code string, then it

may not work out if you jump to a random place in that shell code string, because it may not set the attack up correctly.

But maybe this stuff that your spewing to the heap, is basically just a ton of NOPs and then at the very, very end you have the shell code, right. This is actually quite clever right, because this means that now you can actually goof up the exact place where you jump. If you jump into another one of these NOP things just go boom, boom, boom, boom, boom, boom, boom, then you hit the shell code, right.

So it's like these are the people that you probably see on the team. They're inventing these types of things, right. This is a problem. So that's another way to get around some of this randomization stuff, just by making your codes randomization resilient, if that makes sense.

OK so that's basically a discussion of some of the types of randomness you can use. There's also some wacky ideas that people have had too. So now you know that when you want to make a system call for example, you use this syscall libc function and you basically pass any unique number that represents the system call that you want to make, right. So maybe four is seven and maybe sleep is eight, or something like that, right.

So what that means is that if the attacker can somehow figure out the address of that syscall instruction and jump to it somehow, he or she can actually just supply the system call number that they want to invoke directly, right. So you could imagine that each time the program runs, you actually create a dynamic assignment of syscall numbers to actual syscalls, right. To make it harder for the attacker to get stuff.

There's even some very avant garde proposals to change the hardware such that the hardware actually contains an xor key, that is used to dynamically xor instructions, right. Imagine every time you compile the program, all of the instruction codes that's the xor of some key, right.

That key is put into that hardware register when you initially load the program and

then whenever you execute an instruction, the hardware automatically xor's it, before you continue executing that instruction, right. So what's nice about that is, that now even if the attacker can generate the shell code, the attacker doesn't know that key, right. So it's very difficult for the attacker to figure out what exactly to put into memory.

AUDIENCE: But if he can get the code and he also instructions he can xor it back to the instruction [INAUDIBLE].

PROFESSOR;Oh yeah. This is always a canonical problem, right. So it's like what if someone does this? So that's exactly right. So this is somewhat similar to what happens in the BROP attack where, essentially we've sort of randomized where locations are, but the attacker can do probes, right. And figure out what's going on.

So you can imagine too that for example, if the attacker knows some sub-sequence of code that he's expects to be in the binary, you could imagine just sort of trying to xor the binary with that known code, trying to extract the key. And there's a lot evil in the world, so you're exactly correct about that.

OK so that's essentially the discussion of all the randomization attacks that I want to discuss today. So one thing to talk about before we get to some of the return oriented programming stuff, is you might wonder which ones of these defenses are actually used in practice. And so as it turns out, both GCC and Visual Studio, they both enable stack canaries by default, right. So that's very popular, that's a very well known community.

If you look Linux and Windows they can also do things like non-executable memory. They can also do things like randomize the address space, so that's also [INAUDIBLE]. The baggy bounds stuff however, is not as popular. And that's because of some of these costs that we talked about over here, in terms of memory overhead, CPU, the false alarms, and so on ans so forth. So that's basically a survey of the state of the art, in trying to prevent these buffer overflows.

So now we're going to talk about this return oriented programing stuff. So what I'm

described to you so far today in terms of the address space randomization and the data execution prevention, that's the-- the read, write and execute that I just described.

Those things are actually very, very powerful, right. Because the randomization prevents the attacker from actually understanding where our hard coded addresses are. And the data execution prevention says, even if you can put shell code into the stack, then the attacker can't just jump to it and execute it, right. So at its space, that seems like, man you've really made a lot of progress for stopping these attackers, but of course there are these hackers out there who spend all their time thinking about how to ruin our lives.

So what's the insight behind return oriented programming? The insight is that, what if instead of the attacker being able to generate just new code at attack time, what if the attacker could string together preexisting pieces of code and then string them together in deviant ways, right? And we know that the program contains a ton of code already, right.

So hopefully, or unhopefully, depending on which side of this you're on. If you can find enough interesting code snippets, you can string them together to basically form like this Turing complete language, where the attacker can essentially do whatever the attacker wants to do.

So that's the insight behind return oriented programming. So understand how this works. Let's look at a very simple example that will initially start off very familiar, right. But then it's very quickly going to descend into madness.

So let's say that we have the following program. So we have some program-- sorry, some function and conveniently for the attacker, it has this nice function here called run shell. So this is just going to call system, it's going to execute bin slash bash and then be done.

And then we've got the canonical buffer overflow process or sorry, function down here, basically this thing is going to declare a buffer, and then it's going to use one

of these unsafe functions to fill in bytes in the buffer.

OK so, we know that this can be overflowing OK, this is old news. Now what's interesting is that we have this function up here, run shell, but it doesn't quite seem to be accessed in some direct way based on this buffer overflow. So how can the attacker invoke this run shell command here?

Well the attack may do a couple things. So first of all, the attacker can disassemble the program, run GDB, find out the address of this thing in the executable, right. So you all should be very familiar with doing those kinds of things through the lab. That's the first thing the attacker can do. And then during the buffer overflow, the attacker can essentially take that address, put it in the buffer overflow that's generated and make sure that the function returns to run shell.

So just to make that clear, let's draw that over here. So, you have a setup that looks something like this at the bottom, we have the buffer, that's being overflowed. And then up here, we have the save the break pointer, up here we have the return address, for process message.

So remember that the new stack pointer will be here initially, when we start executing the function. This is the new break pointer, this is what the stack pointer used to be. And then we've got some break pointer up here, for the previous frame. OK, so this should look pretty familiar.

So basically, in the attack, like I said, we've used GDB to figure out what the address is of run shell. So in the buffer overflow, we can essentially just put the address of run shell right here, right. So this is a actually pretty straightforward extension of what we already know how to do, right.

So basically it's saying, if we conveniently have a command that runs a shell, if we can disassemble the binary, figure out where that address is, we can just put that in this overflow array that we have here. So that should be pretty straightforward. Does that make sense?

OK. So, this was a very childish example, because the programmer for some crazy

reason has put this low hanging fruit here. So as an attacker this is like Christmas coming early, right. Now it may not be the case that you have something as delightful as this. So what you could have instead, is something like this.

So let's say that instead of this thing being called run shell, we call it run boring and then maybe this thing just executes bin slash OS let's say. But let's say that everything is not completely lost, because we actually have a string up here that conveniently gives us the path of that.

So what's interesting about this is, that we can disassemble the program, find the location of run boring, but as an attacker, who wants to run an OS? Right, that's no fun But we do actually have a string in memory that points to the path of the shell and actually we also know something interesting too. Which is that even though the program isn't calling system with the argument that we want, it is calling system somehow.

So we know that system must be getting linked into this program somehow, right. So we can actually leverage those two things, to actually call system with this argument here. So the first thing that we do, is we can go into GDB and we can actually figure out where this thing is located in the process binary image, right.

So you just go to GDB, just type in basically print system and I'll give you some information about the offset of that OK. So that's pretty straightforward. You can also do the same thing with bash path. Right, you just use GDB to figure out where this thing lives. It's statically declared string, right. So you'd be able to find out where that lives.

And so once you've done that, now you got to do something a little bit different, right. Because now we actually have to figure out somehow, how to invoke system with an argument of our choosing, right. And so the way that we do that is by essentially faking a calling frame for system.

OK, so remember that a frame is the thing that the compiler and the hardware work together to use to implement the call stack, right. So here's basically what we want

to do. We want to set up something like this on the stack, right. So basically we're going to fake what system would expect to be on the stack, but right before it actually executes its code.

So up here we had the argument of the system, this is the string that we actually want to execute. And then down here, this is where system should return to, when it's done. Right, so this is what system expects the stack to look like, right before it starts execution. It's going to say this is where I should go when I'm finished, this is the thing I should consume as my argument, right. In the past we've been assuming that there were no arguments when you pass the functions, but now this is a little bit different, right.

So basically we just have to ensure that this thing is in that overflow code that we create, right. We just have to make sure this fake calling frame is in that array. So basically the way this will work is, we will do the following.

So once again remember the overflow goes up here. So first, we're going to put the address of system here. And then here, we're going to put just some junk return address. Right, this is where system's going to return after it's finished. For the purposes of the discussion now, we don't care what this does, we'll just make this be just some random set of bytes. And then up here, we're actually going to put the address of bash path, right. So what's going to happen now when we do this buffer in overflow?

So what's going to happen is that, process message is going to finish, it's going to say, OK hey, here's where I should return to, right. And then it's going to pop the stack, right, and now the system code is executing, right. The system code now sees that fake call frame that we created, right. As far as system is concerned, nothing chicanerous has taken place, right. System will say, aha here's the argument that I want to execute, it's bin slash bash, it's going to execute it and voila, the attacker has a shell, right.

So does this make sense? So basically what we've done is, we've now taken advantage of knowledge of the calling convention, for the platform to create fake

stack frames, or fake calling frames I should say. And using those fake calling frames, we can actually execute any function that is already linked and defined in the application. Does that make sense?

OK so, another question you might have is, what if this string wasn't actually in the program? Now to be clear, this string is almost certainly in the program. So that's one funny thing about security, there's just all kinds of fun strings that are laying around, you can just go to town all day long. Well let's suppose we live in bizarro world and like this string is not in the program. So does anyone have any ideas about what we could do to get that string to be in the program?

AUDIENCE: We can put the string on the [INAUDIBLE].

PROFESSOR; Yes exactly, don't trust that man. That's what you can do, exactly. So, what you can do is actually for here, have the address of bash path, actually point here, right. And then you'd put-- up here you would put slash B-I-N slash P-A-T slash 0. So that's how you can get around-- I think I got that math right, because each one of these is 4 bytes. But anyway, so you have the pointer go up here and then boom, you're done.

So now you can actually conjure up arguments just by putting them in the shell code. Pretty horrifying. So this is all building up towards the full BROP attack, right. But before you can mention the full BROP attack, you've got to understand how you just chain together these preexisting things in the code.

So one thing to note is that when I'm setting up this return address here, I just said, eh just put some junk here, it doesn't really matter, we just want to get a shell. But if you're the attacker, you could actually set this return address to something that's actually useful, right. And if you did that, you could actually string together several functions, several function indications in a row, right. That's actually very, very powerful, right.

Because in particular, if we literally just set this return address to jump, I mean it may be that when we ret from it, like the program crashes on it, maybe we don't

want that, right. So can actually start chaining some of these things to do interesting stuff.

So let's say that our goal is that we want to call system an arbitrary number of times. We don't just want to do it one time, we're going to it a arbitrary number of times. So how can we do that? Well, we're going to use two pieces of information that we already know how to get, right. We already know how to get the address of system, right. We just look in GDB and find it. We also know how to find the address of that string, bin flash bash.

Now to actually make this attack work using multiple calls to system, we're going to have to use gadgets, right. This is getting us closer to what's taking place in the BROP paper. So what we need to do now, is find the address of these two Op codes.

Right, so what is this, so this is pop in EAX, so what does this do? This just takes the top of the stack and then it puts it into the EAX register. And what is the ret instruction going to do? It just pops the top of the stack and then puts it into EIP, instruction pointer. OK, so this is what's known as a gadget, right. This is like a small set of assembly instructions that the attacker can use to build these larger more grandiose attacks.

OK, So how can we find this gadget, right. There's actually like some off the shelf tools that hackers use to find these things, it's not hard to get the binary, right. Essentially just do a [INAUDIBLE] for these types of things, right. So it's just as easy to find one of these gadgets, assuming that you've got a copy of the binary and we're not worried about randomization yet. It's very easy to find these things. Just like it's very easy to find the address of system and stuff like that.

So if we've got one of these gadgets, what can we use this gadget to do? Well of course the answer is evil. So, what we can do is the following.

Let's say that we changed the stack so that it looks like this. So the exploit goes this way. And so let's say, we do this. So the first thing we're going to put here is the

address of system. And the thing we're going to put up here, is the address of the pop ret gadget. Then up here, we're going to put the address of bash path and then we're going to repeat this pattern. So we're going to put the address of system, the address of the pop ret gadget, and then the address of bash path.

OK, so what's going to happen here now? Now this is going to be a bit tricky, and these lecture notes are going to be up on the web, so you may just want to listen to what's happening, but this-- when I first understood this, this was like understanding that Santa Claus wasn't real right. So what will happen is-- and by the way, Santa Claus isn't real, I hope I didn't ruin it for everyone.

So what's going to happen? So [INAUDIBLE] is what puts this in memory. So we're going to start here, OK. So what's going to happen? We're going to return to system, the ret instruction is going to pop an entry off the stack, now the top of the stack pointers here.

OK, so system is going to find its argument here, it's going to execute the shell. Then it's going to finish and return to whatever's here, which is the pop gadget. In executing that return, we change the top of the stack pointer to be here. OK, now we are in the pop ret gadget.

OK, so what is that pop ret gadget going to do? It's going to pop what's on the stack, which is this, OK. So now the top of the stack is here. Then we're now in the ret instruction from the pop ret gadget. What's this going to do? Aha, it's going to call system again, right.

So once again the ret is going to pop this off this stack, we are now in system. Top of the stack is here, system is going-- this will trigger calling frame, the system. System takes the bash path argument here. OK, and then it is going to ret, right. Where's it going to ret to? The pop ret gadget again.

So the ret pops the stack, we are now in the pop ret gadget, the ret gadget-- sorry, the pop ret gadget is going to pop this, so on and so forth. OK? So clearly we can chain this sequence to execute an arbitrary number of things, right. And so this in

essence is starting to get to the core of what return oriented programming is.

Note that we have not executed anything in the stack, right. This is what has allowed us to get beyond those data execution prevention bits, right. Nothing's been executed here. We're just sort of jumping to things in unexpected ways to do what we want to do.

OK so this is actually very, very, very, clever. And so what's interesting is that at a high level you can think about us, we've now defined this new model for computation, right. So in a traditional, non-malicious program, you have the instruction pointer that points to some linear sequence of instructions. And you increment the instruction pointer to figure out what's the next thing to do. In essence, what return oriented programing does is, it uses the stack pointer as the instruction pointer.

Right, so as we move the stack pointer, we're pointing to like other blocks of code that we're going to execute. And then at the end of the gadget, you return back to the stack pointer which is then going to tell us the next block of code to execute. OK so that does that make sense? So that's basically how you can avoid the data execution prevention stuff.

That's how you can get around having this no execute bit on pages. So the next thing that we might want to do is defeat stack canaries. So if you remember, this canary was this value that we were going to place on the stack, right. So you can imagine the canary would go right here for example, or right here, and it would prevent someone from overriding the return address, without also overwriting the canary.

With the intuition being that before the system actually jumps to the ret address, it can check to see if the canary has been changed in a way that's incorrect. So that's how the canary works, but can we get around the canary? Can we guess the canary somehow? Well we can actually, if we make a few assumptions about how the system works.

So, how do we defeat those canaries? So the first thing that we want to assume is, that the server, it has to have a buffer overflow vulnerability.

The second thing that we're going to assume, is that the server is going to crash and respond, just restart, if we set the canary value to a bad one.

And the third thing that we're going to assume is that, after the restart, that the canary and any address space randomization that you're doing, is not rerandomized.

Right, so what that means is that, we're going to assume that if we can somehow crash the server, then when the server restarts, it's going to have the same value for the canary. And it's going to have the same locations for all the quote unquote "randomized" stack, heap and code information that it has.

So you might wonder why would this be the case? Why would it be that when the server comes back it doesn't have new locations for things? The reason is because a lot of servers are written to use fork, to create new processes. And if you remember, fork actually inherits-- the child inherits the address space of the address space layout right of the parent, right. This is copy on write pages that change stuff as the child updates things, but if you use fork here, instead of execing a whole new process, any time that parent server process forms new children, those children will have the same values of the canary in the address base, OK. So these are the assumptions that we're going to make to try to defeat these canaries here.

So how can we defeat the canary? Well the attack is actually fairly straightforward. So imagine that the stack is going up this way, right. Imagine you got the buffer overflow here, then imagine that the canary is up here, right. And the canary actually has multiple bytes, right. So what you can actually do is, you can probe those bytes one by one and start guessing values of what those bytes are, right.

So let's say that-- so the canary looks like this. Here's the overflowing buffer, and you want to guess what these bytes are. So the first thing that you guess is you take

your overflow, just to this first byte of the canary and you say, hey, is that byte 0? You write a 0 there, with your overflow. You're either correct or you're incorrect.

If you are incorrect, then the server's going to crash, right. If you are correct you say, aha I actually know the first byte of the canary now, right. Then you start guessing here. You say, are you 0? Probably not, it's going to crash. Are you one? And maybe not, it's going to crash. Are you two? Aha, it doesn't crash, right. So now you've actually found the value of that second canary byte, right.

As you can imagine, you step up this way, and you eventually find all the values for the canary. So once again we're taking advantage of the fact, that crashes are a signal to you, the attacker, that you've actually done something wrong, right. And the server is staying up, in other words, that socket connection's staying open, is a signal to you, the attacker, that you've done something right.

AUDIENCE: Maybe I mentioned something basic here like why do you-- if you know how long the canary is, can you just infect directly? Skip that buffer and overflow those-- the one path there the canary? So like [INAUDIBLE] say you can like [INAUDIBLE] the canary--

PROFESSOR: Yeah, yeah you can't-- so that's right if you-- so if you in fact know the exact location of the canary, right. That can sometimes allow you to skip some of these attacks totally. Because then you can just directly write to the return address, let's say, as opposed to doing some of this buffer overflow nonsense.

But in general, if there's some level of randomization here, if you don't quite know where the stack is for example, then it's tricky to do that, right. So basically the way that the attack proceeds is that you don't quite know what's happening, and so you just very slowly creep your way up memory, down the stack, to figure out where these things are.

AUDIENCE: Can the server instead of crashing when it finds strong canaries, keep the socket open and [INAUDIBLE] deprocess and [INAUDIBLE]?

PROFESSOR: Yeah, so we'll discuss at the end of lecture some of the defenses you can have

against this, but one very, abstractly speaking civil defense, is that when the program crashes, you catch the segfault using a signal handler, do not deduce you're own code by the way. But you can do this, right. You catch that segfault and then the signal handler just keeps that process alive for a bit and that will trick the attack into thinking that, oh I won't get that signal back, in other words.

OK so that's basically how you can guess the value for the canary. And note that you can actually use this attack to sort of figure out arbitrary values that are low in the stack, right. Just by iteratively guessing for each byte what it is, and then using that crash indication as a signal of whether you're guess was correct or not. So that's basically how you can defeat these randomized canaries, assuming that after the server restarts, those things are not changed.

And so we've also shown how you can use the gadgets to string together these more elaborate attacks. So what we're going to look at next is a way that you can use all these techniques to defeat data execution prevention, address-based randomization and canaries on a production system.

Now what we're going to do now is, we're actually going to start looking at 64-bit architectures instead of 32-bit architectures. As it turns out for randomization purposes, 64-bit architectures actually give you a lot more randomness to protect yourself against the attacker. So looking at attacks is much more interesting on those systems.

So that's also the type of architecture that's discussed in the BROP paper. They talk about 64-bit machines. So from now on, assume that we're going to talk about the 64-bit architectures. For the purposes of this discussion, the only difference between a 32-bit machine and a 64-bit machine, is that on a 32-bit machine, the arguments are passed on the stack, right.

So here for example, this is like a 32-bit machine we were assuming, so for example, bash path will pass on the stack. On a 64-bit machine, the arguments are passed in registers instead. OK so like when a function starts execution, it's going to look in certain registers to find where the arguments are. OK, make sense? All right.

So start up here. All right, so now we get to the point of today's paper. Which is the blind return oriented programming. So what's the first thing you want to do, if you want to engage in BROP for fun or profit?

So the first thing you have to do is, you have to find what they call a stop gadget. Now a stop gadget-- and remember that when we say gadget, we essentially mean, return addresses, right. A gadget is identified by the return address, by the start address of that sequence of instructions that we want to jump to, right. So what is a stop gadget?

So a stop gadget is essentially a return address to someplace in the code, but if you jump to it, you're going to pause the program, but you're not going to crash it. OK, so that's why it's called a stop gadget. Now what might that stop gadget be?

You might jump to someplace in the code that then calls via the sleep system call for example, or does pause, or something like that. Or maybe somehow the program gets stuck in an infinite loop if you jump to that place. Doesn't really matter why the stop's happening, but you could imagine several scenarios which would cause that stop to take place.

So why is this useful? Well once the attacker has managed to defeat the canaries using that iterative guessing technique I showed you, he can start to overwrite this return address and start probing for these stop gadgets, right. And so note that most of the random addresses you might put there, they'll probably crash the server, right.

Once again, that's the message to you, the attacker, that's an indication that what you found is not a stop gadget, right. Because when the server crashes your sockets-- your socket connection is closed. You as an attacker know, OK that must not have been a stop gadget. Where if you guess something and then you-- that socket still stays open for awhile, you think, aha I found that stop gadget. So that's the basic idea behind step one. You got to find that stop gadget.

Now step two, is that you want to find gadgets that pop stack entries. And so you

basically have to use this sequence of carefully crafted instructions to figure out when we've got one of these stack gadgets. So this sequence is going to consist of a probe address, a stop address, and a crash address.

So the probe address is the thing that we're going to put in the stack. This is going to be the address of a potential stack popping gadget. This stop gadget is going to be what we found in step one. So this is an address of the stop gadget. And then the crash gadget is just going to be the address of nonexecutable code.

So for example, you could just set this to, just the address zero, right. If you do a ret to this and then try to execute code there, this is going to crash your program. So we can basically use these types of addresses to find out where these stack popping gadgets are. So here's a simple example.

So let's write this over here. So let's say we have these two different examples of a probe, a trap, and then a stop, right. So let's assume that we have down here, we're going to probe at some address, doesn't really matter, starts with a four, ends with an eight. That doesn't matter. Over here, let's say that we look at the address that, let's say starts in a four ends in a C.

So we're saying, we're hypothesizing, that maybe one of these two addresses is going to be one of these stack popping gadgets. And then let's say that the trap up here, like I said this is just going to be address zero, and then let's assume that we found some preexisting stop gadget, some addresses start the [INAUDIBLE] doesn't really matter.

And remember this stop gadget, like maybe this address, points to code that does something like sleep 10, or something like that, right. So when I say that we're going to test these sequences, this is the stuff that we're going to push onto the stack, right.

So similar to over there, when we were pushing these gadgets onto the stack, this is the stuff that we're going to push onto the stack, and we're going to see what happens, right. Now let's say that, this code here, points to the following sequence.

We're going to pop some register, let's say racks, and then we're going to return.

So what's going to happen here? Well so when the system jumps this address, the stack pointer's going to move here, OK. Now we're in the middle of this gadget, right. What's the gadget going to do? It's going to pop racks, OK. Top of stack pointer's now here, and it's going to return to whatever's the top of the stack which is the stop gap, right.

So in this case this gadget gets us to here, and the attacker can tell that this is-- this probe address belong to one of these pop stacking things, right. Because the client connection stays open. Now let's say that this gadget here, pointed to something like the following. Maybe it just does like an xor, for example. So it's just going to xor some registers and then it's going to ret.

So what happens if we try to jump to this gadget? Right, note that this does not pop anything off the stack, OK. It just changes the contents of registers. So what's going to happen? So we're going to be here, we're going to jump to the address of this gadget, stack pointer goes here, OK. We're going to xor these two things, right. Stack pointer's not going to change. Then we're going to return to whatever the top of the stack is, which is 0, 0.

This is going to crash. OK, the client connection to the server is going to close, and as a result, the attacker knows that this is not a stack popping gadget. So does that all make sense? And so you can also imagine that you can-- by coming with more baroque series of traps and stop gadgets and stuff like that, you can find things that for example, pop two things off the stack, right. You can just put another one of these trap instructions there, right.

And so then unless the-- unless this gadget pops two things off, you're going to end up in one of these traps and your code execution is going to blow up, right. And so in the paper they discuss like this thing called the BROP gadget, which is sort of like hilariously complex if you're not used to returning to programming.

What I'll show you today is you can actually just use these very simple pop gadgets

to launch the same attack. Then hopefully after you understand this, the BROP gadget will make more sense. But does everyone understand how we can probe for these little gadgets here?

OK. So, once you've got these gadgets, what do you know? Well you found the location of code snippets that allow you to pop stuff up, one thing off the stack. Precisely one thing off the stack, but you don't actually know into what register they're popping it into. You just know that they're getting popped off, right. And you actually need to know what register these gadgets are popping stuff into, because remember, on a 64-bit architecture, the registers control where the arguments are to this function that you want to invoke, right.

So the ultimate goal to keep in mind, is that we want to be able to create some gadgets that allow us to pop values that we put on the stack into certain registers, and eventually we're going to call a system call that's going to allow us to do something evil. OK, so the next thing that we need to do is determine which registers-- so determine which registers the pop gadgets use.

So how are we going to do that? Well basically we can take advantage of the pause system call. OK, so the pause system call, it takes no arguments, right. And that means that it ignores everything in the registers. OK. And essentially, to find the pause instruction what we can do is, we can chain all of these pop gadgets in such a way, that we put all of them on the stack, in between each one of them we put the syscall number for pause, and then we see if we can actually get the program to hang. Let me give you a concrete example of that.

So we'll do something like this. So here for the return address, we'll put the following. So let's say we have one gadget that pops RDI register, then does a ret. And then up here we'll put the syscall number for pause.

And then let's say that we have another gadget that we found, that does a pop into a different register, let's say RSI. And then we'll put the system call number for pause up here again. And we do this for all the gadgets that we've found and then eventually we put the guest address for pause, or sorry for syscall, excuse me.

Once again, remember how you invoke these system calls. So you basically have to put the number of the system call into the RAX register, then you invoke this libc function syscall which is then going to execute the requested system call, OK.

So what's going to happen when we execute this code? Right, so we're going to come here, we're going to jump to the address of this gadget, and note that as an attacker, all that we know is that this gadget here pops something off the stack. We don't know what the register is yet, right. Put it here just to make the [INAUDIBLE], but the attacker doesn't know yet, right.

So if you jump-- or sorry the-- we jump to the gadget, the stack corners now here, what's it going to do? It's going to pop this syscall number for pause, into some register the attacker doesn't know, and then we're going to continue to go up this chain and so on and so forth. And what you'll see is that each one of these gadgets, one of them hopefully will pop the system call number into the appropriate RAX register.

So that by the time we get up to here, I mean we basically polluted all the registers, with the system call number, but hopefully just one of them has to be correct, right. Because if one of our gadgets does this, then by the time we ret to here, we'll get a pause. Once again, that pause acts as a signal to the attacker, OK. Because if this guest address was wrong, then probably the program's going to crash, right.

So what does this phase of the attack let us do? Well we still don't know which gadgets pop into which registers, but we know that one of them is popped into RAX, which is the one we want to control. And for sure we know the address of syscall, right. Because we were able to induce the pause, right.

So once we've done that, right. Once we know for sure where this thing is, the address for syscall, then we can actually just try the gadgets one by one, right. And see which one of them is actually going to induce the pause, right.

So in other words, cut all the middleman here, let's have a stack it looks like this, and then you just immediately jump to syscall. Did that cause the pause or did it

crash? If it crashed, OK we know this gadget, it pops to RDI for example. OK, get rid of that one, right. Try the next gadget, right. Put the guest address-- put the, well it's not guest anymore, put the real address for syscall up here. Were we able to pause the program? Yes? Aha, so we know that pop gadget must pop into RAX. So does that make sense?

AUDIENCE: So the way to guess the address for system call is just blind transfer?

PROFESSOR: Yeah, so there-- so in the paper, they go into some optimizations about how you can work in a PLT and all that kind of stuff. Like I said, I think it's easier to ignore that for a second, and just look toward the simpler thing first, but yeah in a simple attack that I'm describing, yeah you just put some address up here and you just see if you pause. So does that all make sense?

OK so at the end of this we actually know the location of syscall. We know the location of the instruction that does the pop into RAX. Now you can imagine that we also need gadgets that pop into some other registers too. Suffice to say, you can do similar tests, right.

So instead of like pushing a system call number for pause, push it for some other command that now takes in all arguments in RAX and RDI for example, right. Do the same type of test, right. So basically you can leverage the fact that for any particular set of registers that you want to be able to control, there's some system call that will give you a signal as an attacker, that allow you to figure out whether you successfully broke it or not. Right, so at the end of this phase, you basically have the address of syscall and the address of a bunch of gadgets which allow you to pop into arbitrary registers.

OK and so now let's see so, step 4 is going to be to invoke write. Step 4 is invoke the write system call. So to invoke write, we need to have the following gadgets. You need to be able to pop RDI. We need to be able to pop RSI. We need to be able to pop RDX, pop racks, and then invoke syscall, right.

So as it turns out, what are these registers being used for by system call? So this is

the socket, or more generally, the file descriptor that you're going to pass into write. This is the buffer. This is the length of that buffer. This is the syscall number. And it is called syscall. Right, so if we found all these gadgets, then we can actually now control the values that are put into those arguments, that put in those registers, because we just pushed them on the stack, right.

And so for example, what's the socket going to be? For once you're going to have to do a little guessing here, right. can take advantage of the fact that Linux restricts the number of simultaneous open file connections, for a file that's going to be 2024. And also it's supposed to be the lowest one available.

So we do a little bit of guessing here and figure out what that socket is, put it in there. Now interestingly, what are we going to pass into the buff pointer? Right, we're actually going to use the text segment of the program. We're actually going to pass in that the pointer to somewhere in the code of the program. So what's that going to allow us to do?

That's going to allows us to read the binary, out of memory, using the right call to the client socket. So that the attacker can then take that binary, analyze it offline, right. Just use GDB, or whatever, to figure out where everything is located. The attacker knows that now, every time the server crashes, it's going to have the same randomized set of things in it. So now, once the attacker can find out addresses and offsets for stuff, now the attacker can directly attack those gadgets, right. Directly attack other vulnerabilities, figure out how to open up a shell, so on and so forth.

So in other words, at the point you exfiltrated the binary to the attacker, you basically lost. Right, so this is essentially how the BROP attack works. Like I said, in the paper, there's a bunch of optimization, but really you need to understand this stuff, the basic stuff, before that optimization will start to make sense. And so we can talk about the optimization with me offline if you want, or after class.

But to suffice it to say, this is the basics of how you launch that BROP attack. You've got to find the stop gadget, find those gadgets that pop stack entries. Figure out which of those registers those gadgets pop into, and find out how to figure out

where syscall is, and then invoke write by accumulating all that knowledge.

So very quickly, how do you defend against BROP? Well the most obvious thing is you've got to rerandomize, right. So the fact that crashed servers do not respawn, rerandomize versions of themselves, that allows the crash to act as a signal that let's the attacker test various hypotheses about how the programs working.

So one simple defense is to make sure that you do exec when you spawn your process, instead of fork, right. Because when you exec the process, you create totally new randomized layout space, at least on Linux, right. So on Linux, when you compile with this PIE, the Position Independent Executable flag, you only get that randomized address space that's new if you use exec.

So another event you can use is just use Windows, because Windows basically does not have a fork equivalent, right. So hooray for us. So that means that on Windows, whenever you spawn that new server, it's always going to have a new randomized address space.

I think someone over here mentioned something like, what would happen if for example, when the server crashed, it didn't actually close the connection? Right, so you can imagine one thing that when a crash takes place, we somehow catch that fault and then we keep that connection open for a little while to confuse the attacker and remove that signal, that something's gone amiss.

So that's something you definitely do. What's hilarious about that is, that now your BROP attack turns into a denial of service attack. Because now you just got all the potential zombie processes that are sitting around, they segfaulted. They're useless in society, but you can't let them go, because otherwise you're going to delete this information.

Another thing you might think about to, is you could do bounds checking, right. We just talked a bunch about that, right. But in the paper, they casually dismiss this as saying it has up to 2x overhead, so nobody's going to do that, but you could in fact do that. So that's basically how BROP works.

As for the homework question, the homework questions a bit subtle, because the homework question says, what if you use a hash of the current time, right? Get time of day when you restarted the program. Is that sufficient to prevent this type of attack? Well note that, hashing does not magically provide you bits of entropy if the input to the hash is easily guessable, right.

If I know that you're only going to hash one or two things, it doesn't matter if I have like some a jillion bit hash. Doesn't matter. So I can just guess one of those two values and see what is. So the thing to note is that get time of day, actually has much less entropy than you might think. Particularly because the attacker can actually check what time he or she is launching the attack, right. So that's going to actually remove a bunch of entropy from that calculation, right. So there's some subtleties there. What's the server skew in terms of clock or the client and so on and so forth.

The long story short, using a guessable base value, even with guessable just inside of a range, is super useful for the attacker, right. Particularly because the attack-- we can start subverting a bunch of servers in parallel and know that all of them should have fairly similar values, right. This is a high order of bits, right.

So long story short, the answer is that, it's literally better than nothing to randomize, if you use get time of day, but it doesn't actually provide you as much security as you think. And the other lesson too is, that just because you hash something right, that doesn't matter if you're not actually using that hash in a smart way. You have a question?

AUDIENCE: Oh, still when I did the calculations that some [INAUDIBLE] it seems like maybe to be able to get the offset that the [INAUDIBLE] your [INAUDIBLE] start the process to within like 48 milliseconds?

PROFESSOR: Yes and getting the timing right depends on a bunch of different things, right. But you could take advantage of the fact that the attacker can open up a bunch of connections in parallel, and leverage the fact that even if the initial guess a little bit off, you can still launch multiple guesses on what should be very similar canary

values, and do that attack in parallel. But you're right, there's tricky time and issues.