PROFESSOR: All right. Let's get started. So today we're going to talk about capabilities, continue our discussion of how to do privilege separation. And remember last week we talked about how Unix provides some mechanisms for applications to use if they want to privilege separate the application's internal structure.

And today we're going to talk about capabilities, which is a very different way of thinking about privileges that an application might have. And this is why we have actually these two somewhat distinct readings for today, one of which is this confused deputy problem and how to make your privileges much more explicit when you're writing software so that you don't accidentally use the wrong privileges.

And then the second paper is about the system called Capsicum, which is all about sandboxing and running some piece of code with fewer privileges so that it, very much like [INAUDIBLE], if it's compromised, the damage isn't that great.

Now it turns out that the authors of both of these readings really think capabilities are the answer, because they let you manipulate privileges in a rather different way from how Unix, let's say, thinks about privileges.

So to get started, maybe let's look at this confused deputy problem and try to understand what is this problem that Norman Hardy ran into and was so perplexed by. So the paper is written-- well, it was written quite a while ago, and it uses syntax for file names that's a bit surprising. But we can try to at least transcribe his problem into more familiar syntax with Unix-style path names, et cetera.

So as far as I can tell, what is going on in their system is that they had a Fortran compiler, which sort of dates their design at some level, too. But their Fortran compiler lived in /sysx/fort, and they wanted to change this Fortran compiler, so they would keep statistics about what was compiled, what parts of a compiler were particularly expensive presumably, et cetera.

So he wanted to make sure this Fortran compiler would somehow end up writing to this file

/sysx/stat, that it would record information about various invocations of the compiler. And the way they did this is, in their operating system, they had something kind of like the setuid that we talked about in Unix. Except there, they called it the home files license. And what it means is that if you ran /sysx/fort, and this program had this so-called home files license, then this process that you just ran would have extra privileges on being able to write everything in /sysx.

So it would have these extra privileges on everything in /sysx/, basically, star. It could access all those files in addition to anything that it could access because the user ran it, for example. So the particular problem they ran into is that some clever user was able to do this. So they would run the Fortran compiler, and the Fortran compiler would take arguments very much like GCC takes arguments. And they would compile something like foo.f. Here is my Fortran source code. And they'd say, well, put that output -o into /sysx/stat.

Or more damagingly in their case, there was another file in /sysx that was the billing file for all the customers on the system. So you could similarly ask the Fortran compiler to compile the source file and put the output into some special file in /sysx.

And in their case, this actually worked. Even though the user themselves didn't have access to write to this file or directory, because the compiler had this extra privilege-- this home files license, in their case-- it was able to override these files despite that not being really the developer's intention. This make sense? This is the rough problem they ran into?

So who do they blame? What do they think went wrong? Or how would you design it differently to avoid running into such problems? So the thing they sort of think about here, or they talk about in this write up, is that they believe this Fortran compiler should be very careful when it's using its privileges.

Because, at some level, the Fortran compiler has two types of privileges. It has one stemming from the fact the user invoked it, so the user should be able to access the source file, like foo.f. And if it was some other user, maybe it wouldn't be able to access the user source code.

And in other sorts of privileges is from those home files license thing that allows us to write to these special files. And internally, in the source code of the compiler, when they open a file, the compiler should have been very explicit about which of these privileges it wants to exercise when opening a file or performing some privileged operation.

But their compiler was not written in this way. It was just called open, read, write, like any other program would do. And it would implicitly use all the privileges that it has, which combines-- well, in their system design, it was sort of the union of the user privileges and these home files license privileges. That make sense?

So these guys were really interested in fixing this problem. And they were sort of calling this compiler this confused deputy, because it needs to disambiguate these multiple privileges that it has and carefully use them in the right instance.

So I guess one thing we could try to look at is how would we design such a compiler in Unix? So in their system, they had this whole files license thing. Other mechanisms, then they introduced capabilities. We'll talk about them shortly. But could we solve this in a Unix system? Suppose you had to write this Fortran compiler in Unix and write to a special file and avoid this confused deputy problem. What would you do? Any ideas?

I guess you could just declare this a bad plan. Like don't keep statistics. Yeah?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Sure. That could be, right? Well, yeah. So you could not support flags like -o. On the other hand, you might want to allow specifying which source code you want to compile so that maybe you could read the billing file or read the statistics file, which maybe should be secret. Or maybe the source code has-- maybe you can support a the source code on standard, but it has include statements, so it needs to include other pieces of source code. So that's a little tricky.

AUDIENCE: You could split up the application [INAUDIBLE].

PROFESSOR: Yes. So another potentially good design is to split it up, right? And realize that this fort compiler really doesn't need all these two privileges at the same time. So maybe we should have our Unix world /bin/fortcc or something, the compiler, and then this guy is just a regular program with no extra privileges. And then we'll also maybe have a /bin/fortlog, which is going to be a special program with some extra privileges and it'll log some statistics about what's going on in the compiler. And fortcc is going to invoke this guy.

So how do we give this guy extra privileges? Yeah?

AUDIENCE: Well, maybe if you use something like setuid or something, like fortlog, then presumably any other user could also log arbitrary data through it.

PROFESSOR: Yeah. So this is not so great. Because on fortlog, presumably the only way to give extra privileges in Unix is to in fact make it owned by, I don't know, maybe the fort UID, and that's also setuid. So every time you run it, it switches to this Fortran UID. And maybe there's some special stats file. But then in fact anyone can invoke this fortlog thingy. Which is maybe not great. Now anyone can write to the stats file.

But maybe this example is not the biggest security concern about someone corrupting your statistics. But suppose this was a billing file. Then maybe the same problems would be slightly more acute. Yeah?

AUDIENCE: But you can always make your [INAUDIBLE] stats you want, right? Instead of [INAUDIBLE].

PROFESSOR: So in some sense, yeah. If you're willing to live with arbitrary stuff in your statistics or logging file, then maybe that's true.

AUDIENCE: Even if you [INAUDIBLE], you can already make your C code have whatever statistics that you'd want to be recorded.

PROFESSOR: You could. Yeah. Yeah. So it might be that in this case, it doesn't really matter that you can log arbitrary stuff. So that's true. Yeah. So if you cared about who can invoke this fortlog thing, could you really do something about it in Unix, or not so much? Yeah?

AUDIENCE: [INAUDIBLE]. It would make both of them setuid. Now the fortcc would read that source files. It would switch back to the saved UID, just the user UID. Remote fortlog in a setuid, which has permissions to execute fortlog. And that fortlog would setuid again [INAUDIBLE].

PROFESSOR: Right. Yeah. So there is this rather elaborate mechanism in Unix that we skipped on last Monday's lecture, that actually allows an application to switch between multiple UIDs. if it was setuid to some user ID, then it could say, well, now I want to run with this user ID. Now I want to run with this other user ID. And it could sort of carefully alternate between these. It's a little tricky to do it right, but it's probably doable. So that's one potential design.

I guess another hack you could maybe try to do is make this fortlog binary only executable to a particular group and make fortcc a setgid binary to that group. It's not great, because it obliterates whatever group list the user had initially. But who knows? Maybe that's better than nothing.

Anyway, so it's a fairly tricky problem to solve in an entirely satisfactory fashion with these Unix mechanisms. Although, maybe you should rethink your problem and not worry about your statistics file as much in the first place. But how do we think about what's going wrong in the design?

Well, there's two things we could try to learn from this, or basically, what went wrong. And one interpretation that one party wants us to take away is this notion that he calls ambient authority. So what is ambient authority? Can anyone figure out what they meant? They've never exactly defined this. Yeah?

AUDIENCE: It means you have the authority given to you by the environment. So as if [INAUDIBLE] user with no limitations.

PROFESSOR: Yeah. So you're making an operation, and you can specify what operation you want. But the decision of whether that operation is going to succeed comes from some extra implicit parameters in your process, for example. And in Unix, you can figure out what this ambient authority check might look like.

So if you make a system call, then you probably supplied some sort of a name to a system call. And inside of the kernel, this gets mapped to some sort of an object. And the object presumably has some kind of an access control list on it, like the permissions on a file, et cetera. So there are some permissions that you can get from the object. And that should decide whether an operation is going to be allowed on this name of the application supplied. This is sort of what the application gets to see.

Inside of the kernel, there's also the current user ID of the process making the calls. So this is the current prox UID. And this thing goes into the decision of whether to allow a particular operation or not. So it's the current process user ID that's this ambient privilege. Whatever operation you're going to try to do, the kernel will actually try, in some sense, as hard as possible to allow it by using your current UID, and your current GID and whatever other extra privileges you might have.

And as long as there's some set of privileges that allow you to do it, it'll let you do it. Which is maybe not the best thing to do if you aren't fully aware of what all these problems are. Maybe you don't want to use all of them to open a particular file or make some other operation.

Does this make sense, roughly what ambient privilege is? In the case of an operating system, it basically ends up being the fact that a process has some sort of a user ID. Are there non-OS examples of ambient privilege you guys can think of? Like when you're making an operation, something about the identity of the caller, the terms of whether they'll succeed or not.

Like one example is probably firewalls, as well. So this is just an OS example. And in privilege, another is the firewalls on the network. Because any operation you do from a machine inside of a firewall is going to be allowed because, well, you just have that IP address, or you're on that side of a network. And if you're outside, the same operation will be disallowed. So it's also a solar problem.

Say you visit some website, and the website includes a link to some different server, well, maybe you don't want to use the privileges that you have or the inside of your network to access that link. Because maybe it'll access your internal printer and exploit it in some way. And really, the guy that provided you the link shouldn't have been able to reach the printer in the first place, because they were on the outside. Or a firewall that your browser, maybe by visiting uplink, will be tricked into doing this.

It's sort of a moral equivalent of this confused problem on the network models. Yeah?

AUDIENCE: [INAUDIBLE] permission are directly affected also.

PROFESSOR: Yeah.

AUDIENCE: Because it's essentially DAC, potentially, in the Capsicum.

PROFESSOR: Yeah. So this is pretty much what the Capsicum guys think of as discretionary access control. And the fact that it's discretionary, well, this is not quite what discretionary access control means. But what discretionary access control means is that the user, or the owner of an object, can decide what security policy will look like to an object. Which seems very natural in a Unix setting. it's my files, I can decide what I want. I can give them to you, or I can keep them private, great.

So almost all DAC systems do look like this, because they want to have some sort of permissions that a user could modify to control the security policy for their files. The flip side is mandatory access control. We'll talk about it in a little while, but at some level, they have this very philosophically different view of the world. They think, well, you're the user. But someone else will set the security policy for how you use this computer.

And this sort of came out of the military in the '70s or '80s, when they really wanted to have classified computer systems where, well, you're working on some stuff and it's marked secret. I'm working on some stuff that's marked top secret. So my stuff just can't go to you. It's not up to me whether to set the permissions on a file, et cetera. It's just not allowed by some guy in charge.

So mandatory access control is really trying to enforce these different kinds of policies in the first place, where there's the user and the application developer. And then there's some guy separate from the user and the developer that sets the policy.

And, as you can sort of guess, it doesn't always work out. Well, we'll talk about it in a bit. But that's what discretionary versus mandatory means at this control.

All right. So there's many other examples that you could imagine where we have ambient authority. And it's not inherently bad, law but it's just something that you have to be very careful about. If you have a system with ambient authority, you should probably be very careful if you're performing privileged operations. You should make sure that you're really using the right authority and not accidentally being tricked very much like this Fortran compiler 20 years ago. 25 now.

All right. So this is one interpretation of what goes wrong. And it's not necessarily the only way to think about what goes wrong, right? Another possibility is that, well, wouldn't it be nice if it was easy for an application to tell whether it should access a file on behalf of some principle?

So maybe another problem is that the access control checks are complicated. So in some sense, when the Fortran compiler is running, and it's opening a file on behalf of a user, it basically needs to replicate the same exact logic we see drawn out here, except that the Fortran compiler needs to plug-in something else here. Instead of using its current privileges, and all of them, it should just replicate this check and try to make it with a different set of privileges.

So in Unix, this turns out to be fairly tricky to do, because there's many places where these security checks happen. if you have symbolic links, then the symbolic link gets looked up, and that path name also gets evaluated with someone's privileges, et cetera.

But it might be that, in some system, you could simplify this access control check, where you could do it yourself in an application. Does that seem like a reasonable plan to you guys?

Would you go with that? Any dangers of replicating these checks? Yeah?

AUDIENCE: Well, if you do the checks in the application, you could just not do the checks.

PROFESSOR: Yeah. So you could easily miss the checks. That's absolutely right. So in some sense, what the Fortran compiler did here, well, they didn't even bother trying to do the checks, now that they screwed them up.

Another possibility, in addition to missing the checks, is maybe the kernel will change over time, and it will have slightly different checks. It will introduce some extra security measure, and the application will be left behind. And it will implement old style checks. And probably not a great plan.

So recall, one good idea in security is to have economy of mechanisms. So there's only a small number of places that are enforcing your security policies. You probably don't want to replicate the same functionality in applications in the kernel, et cetera. You really want to boil it down to one place. That roughly makes sense? OK.

So what is this capability, I guess, idea where thinking might solve this authority problem? Well, there's some formal definition for the thing. But really, you can get very close by thinking of Unix file descriptors as a capability.

So I guess the alternative to this picture, in capability world, is that instead of having the application supply name, and you look up an object, you get a permission, you decide whether to allow it based on some ambient authority, instead, the capability is the picture looks very simple. You have a capability, and if you have a capability, it points to an object. And maybe the capability has some small number of restrictions of what you can do with an object.

But basically, if you have the capability to an object, you can access the object. It's actually very simple. So there's no ambient authority that decides whether an operation on a capability is going to be allowed. The only thing is that maybe the capability has a couple of extra bits, or this mass that they described in the paper, which says, well, you have a capability for this file, as it's restricted to read operations only. Or it's restricted to write or append operations only.

And then your security decisions are all of a sudden very easy. Because if you have a capability, you can do something. If you don't, you can't. Make sense?

So I guess one important property of capability is that they should actually be unforgeable, as the papers talk about. So what does it mean to be unforgeable, or why do we want this in this capability world?

Well, I guess this actually may be almost too obvious here. Well, if you can make up any capability you want-- I can make up a capability for any of your guys' files and go access it. So if I can make it up, and I'll access it, and there's nothing else in the security design, that stops me from accessing an object once I can manufacture a capability.

So it's important that these capabilities can't be made up out of thin air by the application or by whatever's running. How is this getting forced, if we think of file descriptors as capabilities? So many of you guys actually submitted this as the big question about Capsicum. What do you think? What prevents an application from synthesizing a capability in this file descriptor world?

Could you synthesize a capability? Yeah?

AUDIENCE: Well, it was probably like a structure and a construct that says that they have a capability for certain file descriptors.

PROFESSOR: Yeah. So it's actually fairly easy to see what goes on once you look at what exactly is a file descriptor, right? So a file descriptor is basically just some sort of an integer. And this integer-- like in Unix, you have file descriptor 0, which refers to your input, file descriptor 1 which refers to your output. Rockwell

But really, these are just integers in user space. And this is what the application can presumably do, and it can choose any integer it wants. But whenever you try to do something to a file descriptor, which is one of these integers, the kernel will always interpret the integer according to your current process's file descriptor table.

So for every PID-- let's say, well, this is PID, I don't know, 57. So I'm process running. It has an open file table, and each integer from supply from user space, refers to some entry in this table. And of course, the kernel should check that the integer is in bounds in this stable. It isn't negative. It doesn't go past the end of the table. Otherwise, it will have the usual buffer overflow problems, et cetera.

But if you carefully check that the integer is in bounds in the kernel implementation, then the only possible things that the application can refer to by a file descriptor are entries in this table. So presumably, the kernel will somehow make sure that you legitimately guard a particular capability.

So when you, for example, open a file outside of this capability model in Unix, well, the kernel, after the open call succeeds, it's going to change that file descriptor table entry to point to a particular open file, like maybe open/etc/pwd.

And now, the entry at this slot on the table points to an open file. Some of them might actually be null. Maybe you don't have an open file with a particular index in this table. And as a result, what does it mean to forge a capability? The only thing you can do in user space is make up an integer. And the only integers that would make sense to make up would be entries that point to non-null entries in this table. And those guys are exactly the capabilities that you have.

So does that make sense why it's difficult, in this file descriptor world, to actually forge capabilities in the first place? So it's kind of cool, right? Like the only files that you have opened are exactly the things you can operate on. And there's nothing else that you can potentially touch and effect. Make sense? Any questions?

All right. OK. So I guess, how would capabilities help solve the ambient authority problem that Norman Hardy is excited about with his Fortran compiler? So what would be the file descriptor moral equivalent solution to this sysx/fort thing? Do they actually solve the problem? Yeah?

AUDIENCE: Well, they just use the appropriate capabilities whenever they're needed. So when you have to access the output file, in the statistics, you use the capability [INAUDIBLE] file. But when you're accessing the file you're about to read, you don't use that capability.

PROFESSOR: Yeah. So I guess really what it boils down to is that somehow the Fortran compiler should just already have a file descriptor open for that /sysx/stat file. So they don't really describe, in their short paper, about how we don't get that capability.

But it basically means you shouldn't really pass file names around. You shouldn't set past file descriptors. So you could actually come up with a perhaps much more elegant design for our Unix replacement on the Fortran compiler using capabilities. So maybe the plan is we should just have a Fortran compiler front end that doesn't have any extra privileges, and it takes all these arguments you give it, and converts all the path names you supply to it into open file descriptors.

So the alternative design I am thinking of here is that maybe we'd have a program fort1, which is the front end. And it would take some sort of a file, foo.f, and all the other arguments, -o, whatever. And it doesn't actually implement any of the compiler logic, anything else. All it looks for is path names in its arguments, and it's going to open them and establish file descriptors for them.

And the cool thing is that, because it has no extra privileges, if the user can't have access to some file name, then it will fail. Those are great. And then once this front end has opened all these file descriptors, it can execute some privileged extra component, like the actual setuid Fortran compiler. So maybe then it'll run fort. This guy's maybe setuid to some special user ID that has access to the stats file. But it doesn't actually accept any path names as input. All it's going to do is accept file descriptors.

And, in that case, the file descriptor is already prove that the caller had access to open them. Does the property make sense? So it of course doesn't solve every issue. I'm just sort of sketching out how capabilities might help. But that's roughly the plan, is that you should demonstrate the fact that you have access to a particular name by just opening it and passing a capability, instead of saying, why didn't you try to open this file and maybe accidentally use some extra privileges. Yes.

AUDIENCE: So does this generalize to having one process per capability?

PROFESSOR: Does this generalize? Well, of course you can have as many processes as you want. You can have multiple processes per capability, but I'm not sure--

AUDIENCE: [INAUDIBLE].

PROFESSOR: I'm still not sure what you mean by one property.

AUDIENCE: So we have [INAUDIBLE] capabilities the user has.

PROFESSOR: That's right.

AUDIENCE: And then we have the fort.s access to this past file.

PROFESSOR: That's right. Yeah. So the way to think of it is, you don't necessarily need a separate process for every capability. Because here, the fort1 thing might open many files and might pass many capabilities to the privileged fort component. The problem here-- the reason that this might seem like you want a separate process for every capability is that we're sort of dealing with this weird interface between capabilities and ambient privileges.

Because fort1 sort of does have ambient privilege. And what we're doing is basically we're converting this ambient privilege into capabilities in this fort1 process. So if you have multiple different kinds of ambient privilege, or multiple different privileges that you want to carefully use, then maybe what you want is a separate process holding that privilege. And whenever you want to use a particular set of privileges, you'll ask the corresponding process to please perform a separation. And if it succeeds, give me back the capability.

So that's maybe one way to think of this. There's been actually some operating system designs that are entirely capability-based, there are no ambient privileges whatsoever. And it's kind of cool. Unfortunately, it's more of sort of an interesting reading experience. Like oh, yeah, you can do it. That's pretty cool. But it's probably not really practical to use in a real system, unfortunately. It turns out that you really do want not so much ambient privilege but being able to name an object and tell someone about an object without conveying necessarily the rights to that object.

So maybe I don't know what privileges you might have over some shared document, but I do want to tell you, hey, well, there's a shared document. If you can read it, read it. If you write it, great, write it. But I don't want to necessarily convey any rights. I just want to tell you, hey, there's this thing, go try it. So it's a bit of a bummer in a capability world that it really forces you to never talk about objects without conveying rights to that object.

So it's an important idea to know about, and to use it in some parts of a system, but probably not the be all end all solution to security, much like almost anything else [INAUDIBLE] about here. Make sense? Yeah?

AUDIENCE: So if the process has capabilities given to it by some other process, and it happens to already have the capability to that object, that's greater. Can it compare them to make sure that they're about the same object? Or will it just use the one that's greater?

PROFESSOR: So the thing is that a process doesn't implicitly use the capabilities. So that's the cool thing about capabilities. You have to explicitly name which one you're using. So think of it in terms of file descriptors. Suppose that I give you an open file descriptor for some file, and it's read only. And then someone else gives you another capability for some other-- maybe the same filem maybe a different file, and it's read/write.

It's not all of a sudden that if you're trying to write to the first file descriptor you had that was read only, all of a sudden those will start succeeding because you have this extra writeable file descriptor open. So that's sort of the cool thing. You don't want this ambient privilege. Because if you think of these capabilities as a bunch of privileges that just keep accumulating in your process, then you'll actually just end up with ambient privilege again. You just have all these magic capabilities, and people have actually built such libraries. Basically, well, they manage your capabilities for you. They sort of collect them. And when you try to perform an operation, they look for the capabilities and find the one that'll make it work.

That exactly brings you back to this ambient authority that you were trying to avoid. So the cool thing about capabilities is that it's almost like a programming construct, where it makes it easy for you-- which is a rare thing in security-- it makes it easier for you to write code that specifies exactly what privileges you want to do from a security standpoint. And it's actually a fairly natural code to write.

So if you get into that mindset of always carrying around this privilege with the object you're accessing, it seems like a cool thing to do. It doesn't always make sense, but sometimes it does. Any other questions? OK.

So that's more on the ambient authority that we've look at here. It turns out that capabilities are also great for other problems, as well. And in particular, the problem of managing privileges often shows up when you want to run some untrustworthy code. Because you want to really control which privileges you give it, because you think it will misuse any privileges you give it at all.

And this is the slightly different point of view from which the authors of the Capsicum paper are coming at capabilities. So they're of course clearly aware of this ambient authority problem, but it's sort of a different problem that you might or might not care about solving. But the particular thing they really care about is they have a really large privileged application, and they worry that there's going to be bugs in different parts of that application source code. So they would like to reduce the privileges of different components of that application.

So in that sense, the story is very similar to OKWS. So you have-- for sandboxing, you have some large application, you break it up into components, and you will limit what privileges each component has. So where does this make sense? Like OKWS is clearly one example. What are other situations where you might care about prileged separation? Well, I guess in the paper they describe the examples I actually got to run.

So things like tcpdump and other applications that parse network data. So why do they worry so much about applications that parse network inputs? What goes wrong in tcpdump? Why are they so paranoid?

AUDIENCE: Well, an attacker can control what's being sent and what's being called.

PROFESSOR: Yeah. I think what they really worry about is, very much like with OKWS, they worry about that attack surface and how much can an attacker really control the inputs? And with these network parsing programs, there's a lot of control that that factor has. They have the exact packet.

And the reason that this was so problematic is that if you're writing code in C that has to parse data structures, you're presumably going to do lots of pointer manipulations, copying bites into arrays, allocating memory. And as you are now experts, this is super fragile. And you can easily have memory management errors that lead to pretty disastrous consequences. So this is the reason why they're very excited about sandboxing various network protocol, parsing things, et cetera.

Another probably real world instance where you really care about this is in your browser. You probably want to sandbox your Flash plug-in, or your Java extension, or whatnot. Because they're pretty large attack surfaces as well that have gotten exploited pretty aggressively.

So it seems like a reasonable plan. Like if you're writing some piece of software, you want to sandbox different components of it. What about more generally, if you download something from the internet, and you want to run it with fewer privileges? Is this sort of Capsicum style isolation a good plan for that? I could download some random screensaver or some game from the internet. And I want to run it on my computer, and I want to make sure it doesn't screw up whatever I have laying around.

Would you use Capsicum? Would this be a good plan? Yeah?

AUDIENCE: You could write a sandboxing program, which you'd use Capsicum to sandbox [INAUDIBLE].

PROFESSOR: Right. You could try to use Capsicum. So how would you use Capsicum? Well, you'd just enter into the sandbox mode with cap_enter. And then you run the program. Would you expect it to work? I guess the problem is that if the program wasn't really expecting to be sandboxed with Capsicum, then all of a sudden the program will try to open any simplified-- it'll open a shared library, and it can't open the shared library, because it can't open/liv/ something else. That's not allowed in capability mode.

So it's a bit of a problem. So typically, these sandboxing techniques that we're going to look at here-- capabilities, style, stuff, and so on-- really are best used when the developer is sort of building the application aware that the code is going to run in this mode. There's probably other kinds of sandboxing techniques that could be used for unmodified code, but then the focus, or the requirements, change a bit.

So in Capsicum, they don't really worry about backwards compatibility. Well, we have to open files differently? Sure, we'll open them differently. Whereas, if you want to write existing code, you probably want something more like maybe a full virtual machine. So you could open a VM and run it there. And it's very compatible, and there's no question that it'll just run, and probably not--

Well, it's actually a good thought exercise. Should we use virtual machines to sandbox instead of Capsicum?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. The overheads are probably quite significant. So the memory overhead is pretty bad. It could be. But what if we don't care about memory overhead? So maybe virtual machines gets really good, and they don't actually use that much memory. Is it still a bad plan?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. So it's kind of hard to control what happens on the network, because either you give the virtual machine no access to the network at all, or you connect to a network through NAT mode or something in Preview or VMware. And then it can access the whole internet. So you have to much more explicitly control network by maybe setting up firewall rules for the virtual machine, et cetera. That's maybe not so great.

What if you don't care about network? What if you're some simple video or tcpdump parser. You just spin up a VM. It's going to parse your tcpdump packets and spit you back after your presentation that tcpdump wants to burn to the user. So there's no real network I/O. Maybe you're, for some reason [INAUDIBLE] still?

AUDIENCE: Because the initialization overhead is still large.

PROFESSOR: Yeah. So it's maybe like an initial overhead of starting a VM. So that's true. There's some performance stuff. Yeah.

AUDIENCE: Well, you might want to have database rights and things like that.

PROFESSOR: Yeah. But even more generally, what you're getting at is what if there's a real data that you care about here? And it's really hard to share. So VMs are really a much more sort of separation mechanism, where you can't really share stuff across VMs very easily. So it's good for situations where you have a very isolated program you want to run, you basically don't want to share any files with any directories, any processes, any pipes even. And you just let it run separately.

So it's great. It's probably, in some ways, stronger isolation than what Capsicum provides, because there's probably fewer ways for things to go wrong. And, you know, all these problems we talked about so far. But it's also not applicable in many of the situations where you might want to use Capsicum, because in Capsicum, you can actually share files that have very fine granularity between sandbox [INAUDIBLE] by just giving it capability to [INAUDIBLE] file. This is something that's very easy to do in Capsicum, and would require quite a bit of machinery in a virtual machine setting. That makes sense? Questions? All right.

So does that seem like a useful primitives to have to maybe sandbox stuff. So I guess we're going to talk about different ways to try to sandbox something. And Capsicum in particular is the new thing here that uses capabilities. But just by comparison, I guess, you can do some sandboxing in Unix, as we saw with OKWS. Right? It's just not great from several standpoints.

So let's maybe take the example of tcpdump and see why tcpdump is difficult to sandbox with Unix mechanism. So remember, in the Capsicum paper, these guys took tcpdump. And the way tcpdump works is that it opens some special sockets and then runs basically parsing logic on network packets. And it proceeds and prints them out to the users' terminal.

So what would it take to sandbox tcpdump with Unix primitives? Have you restricted privileges? So I guess the one problem with Unix is that you basically have to-- well, the only way to really change privileges is to change the inputs into the decision function that decides whether you can actually access some object or not. And the only things you can really change are, well, you can change the privilges of the process, which means it sends UID to something else.

Or you could change the permissions on various objects that are laying around in your system. Or probably both, in fact, right? If you wanted to sandbox tcpdump, you'd probably have to pick some extra user ID and switch to that while you're running. Probably not an ideal plan, because you probably don't mean for multiple instances of tcpdump to run as the same user ID.

So if I compromise one instance of tcpdump, it doesn't really mean I want to allow that factor to now control the other instances of tcpdump running on my machine. So that's potentially a bad part of using user IDs here.

Another problem is that, in Unix, you actually have to be root in order to change the user ID of the process or something else, or user privileges or switch them to something else. That's not great either. And another problem is that, regardless of what your user ID is, there could be files that allow access to them. So there could be world writable or world readable files in your file system. Like your etc password file. Regardless of what your UID is, the process will still be able to read that password. So that's not so nice.

So the result, in order to sandbox a unit, you probably have to do both-- some UID changing and maybe careful look at the permissions of all the objects to convince yourself that there's no world writeable file that's really sensitive. Or there's no world readable file that you don't want that hacker to get access to.

And I guess [INAUDIBLE] true that you get another mechanism unit that you can use. But it all starts to add up. If you see it through, then it might be hard to share files or share directories and so on. So does that make sense? Just in terms of contrast for what Capsicum is trying to solve? Any questions about Unix stuff? All right.

So let's look at how Capsicum tries to solve this problem. So in Capsicum, as we keep alluding to, the plan is very much that once you enter the sandboxing mode, everything is going to be accessed only through capability. So if you don't have a capability, you simply cannot access any objects.

So these guys, in the paper, make a huge deal about global namespaces. So what's this thing about a global namespace, and why are they so worried about it? What's an example of a global namespace these guys worry about?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. So a file system from them is sort of the prime example of a global namespace. You can start a slash, and you can basically enumerate any file you could, right? Like go to someone's home directory-- /home/nickolai/ something, something. Why is this bad? Why are they against global namespaces in Capsicum? What do you think? Yeah?

AUDIENCE: Well, if you have the wrong permissions, then use authorities, and then you can get in trouble.

PROFESSOR: Yeah. So the problem is that this is Unix after all. So there are still regular permissions on file. So maybe you really want to sandbox some process and can't read anything at all in the system and can't write to anything. But if you can name a file starting from scratch, you'll find some stupid user that has a world writable file in their home directory. And that would be not so great for the sandboxing client.

And I guess more generally, the way they're thinking of it is that, with capabilities, you could, in principle, enumerate exactly all the objects that a process has. Because you could just enumerate all the capabilities in the file descriptor table, or whatever it is that's storing capabilities for you. And those are the only things that the process could ever touch.

And if you ever have access to our global namespace, and this was potentially unbounded. Because you could-- even if you have some limited set of capabilities, maybe you'll start from slash again and find some new file, and you'll never really know what is the set of operations or objects that a process could access.

So this is the reason they're so worried about global namespaces because it goes against their goal of precisely controlling all the things that a sandbox process should have access to. Make sense? All right.

So they tried to eliminate global namespaces with a bunch of kernel changes to the FreeBSD, in their case, kernel to make sure that all the operations go through some kind of capability, which is, in their case, a file descriptor.

So just to double check, do we really need kernel changes? What if we just do this in a library? So we implement Capsicum, which they already have a library. And all we do is we change all these functions, like open, read, and write, to all very exclusive use capabilities. So all operations will go through some capability, and look it up in the file table, et cetera. Does that work? Yeah?

AUDIENCE: You could always make a sys call.

PROFESSOR: Yeah. So the problem is that there was this existing set of systems calls the kernel will accept. And even if you implement a nice library, it doesn't prevent a bad process or a compromised process from making the sys call directly. And then you have to have the kernel enforce something or other. Yeah?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. So I think it's a question of-- I guess what is your threat model? Exactly. So for the compiler, the threat model is that the programmer is maybe not paying attention a whole lot, but it's not really a compromised compiler process, not an arbitrary code. So if we just help the well-meaning developer do the right thing, then a library will probably suffice.

On the other hand, if we're talking about a process that could be our executing arbitrary code and could be trying to bypass our mechanisms in any possible way, then we have to have a strong enforcement boundary. And a library doesn't provide any kind of strong enforcement guarantees. Whereas a kernel, in our case, would do that.

OK. So what do they actually make in terms of changes to the kernel? So I guess the first thing is this system call that they call cap_enter. And what happens once you run cap_enter? Once you've [INAUDIBLE] cap_enter from your process?

So as far as I can tell, what happens is that the kernel will stop accepting any system calls that refer to global namespaces. And the only thing you'll be able to do is refer to existing file descriptors that you have open in your process. So cap_enter will put your process in a special mode where you cannot use the regular system called open, and instead you have to do things like openat.

So there's this new sort of family of systems called, in Unix like operating systems, where instead of having open take a single path name, you can actually you openat, where you pass it a first argument which is a file descriptor for a directory and the second is some sort of a name. And the open at system call will open this name relative to whatever directory the file descriptor points to.

So this is a much more capability-like version of open, where you can still have file descriptors pointing to directories, but you can-- well, sorry. You can still direct your operation. But in order to do this, you have to have a capability to the directory in the form of an open file descriptor for that [INAUDIBLE]. Make sense? OK.

So do they need any other kernel changes? Is there anything else they worry about? So I guess there's another-- yeah?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. So what do they do about network access, right? So what happens in capability mode?

AUDIENCE: I guess they have capabilities for security packets [INAUDIBLE].

PROFESSOR: Yes. So I think the way they basically do it is that they treat the network as a global namespace, very much like a file system. So I think once you enter capability mode, you cannot create a new socket. Or you cannot create a new socket and connect to some arbitrary machine, or to some arbitrary address or fort number. You have to basically create all the connections you want ahead of time and fill them in as capabilities. Or maybe you'd have to get them from someone that will pass you a file descriptor.

But basically, once you're in capability mode, the set of file descriptors you have open completely enumerates all the machines you'll ever talk to. So you can find open connections. Maybe you're listening on a forge. That's OK. But you cannot connect to an address specified by an absolute name, kind of like a global namespace would allow you to do it. That make sense?

So it's access through the networking namespace, as well. What do they do for processes? So another global namespace, I guess, in Unix, is the the PIDs themselves. So the example of a system call that operates in this name space is "kill." So I could kill PID 25. And I could-- well, presumably I'll put a single number in there, too.

But I could actually kill a process by its PID number. How do they fix this in Capsicum? What's their plan? Yeah?

AUDIENCE: File descriptors with processes.

PROFESSOR: Yeah. It's actually kind of cool. It's like, I wish Unix had this all along. Which is that, instead of having these different kinds of numbers or PIDs, instead, when you fork off a process, actually having new variant of fork called pdfork, or Process Descriptor Fork. And what it does is when it creates a child process, it actually sticks a reference to that child process into your file descriptor table somewhere.

And this is your new process. And you can operate on a child process by specifying the file descriptor number. Well, it would be pretty cool, because you can now pass your child process to someone else and say, well, if you can go and kill them now, or you can manage this process however you want, you'll get notifications when the process dies. It'll look like a readable file descriptor, et cetera.

So they really try to homogenize everything into looking like a file descriptor of some sort here. And with these kernel changes, you can finally have all the functionalities you might care about. You have the support for sockets already, process descriptors, et cetera. And you have a way of constraining what the process can do. Because it cannot refer to any of the global names anymore after [INAUDIBLE].

All right. Any questions? So here's an interesting puzzle. I was trying to understand from the paper. They make a big deal about dot dot in looking up directory names. So they basically say, well, once you're in capability mode, when you pass a particular name to openat, you cannot use dot dot in those names. And presumably, if you have a Simulink, if a Simulink's target contains dot dot, they will reject it if you're in capability mode.

So is this strictly required? Could you imagine a safe design in principle that allows the use of dot dot? Yeah.

AUDIENCE: Well, you'd need to be able to find whether they have a file or a capability that allows the masses to the parent directory.

PROFESSOR: Right.

AUDIENCE: So it's trivial to go down, because any subdirectory-- you already have access to it by having the capability.

PROFESSOR: That's right. Yeah.

AUDIENCE: But going up, you need to see whether you have any capabilities for the parent directory.

PROFESSOR: That's right. Yeah.

AUDIENCE: Search for it somehow.

PROFESSOR: Yeah. So that's a little bit tricky. And also, it goes against the grain of this whole explicit authority thing. What about if you're using dot dot inside sort of a single open call? So for example, what if you call something like openat some particular directory or file descriptor number, and you open something like, I don't know, b/c/../..?

In principle, this might be safe, right? Because you go down some directory, and then you just climb back up out of it. Yeah?

AUDIENCE: What if c is [INAUDIBLE]?

PROFESSOR: Yeah. So it's a little bit tricky, of course, to define exactly what it means to be safe. Right? You probably have to make sure that c isn't a Simulink that goes somewhere else and so on. Yeah. That's a fairly tricky proposition, to get this right. And I think, in the paper, what they basically argue about is that it's actually quite difficult in practice to implement a set of checks that's sufficient and bypasses all the possible rate conditions here. So they basically just do the conservative thing and disallow any dot dot at any time once you're in capability mode.

There's some interesting rate conditions you could come up with here. The lecture notes have more details. But basically I think these guys are being extra cautious in defining what's allowed and what's not allowed in capability mode. OK.

So here, to answer your question, once you enter capability mode, it seems to be all controlled by your file table. Does your UID still matter, once you enter capability mode? [INAUDIBLE]. Yeah?

AUDIENCE: Well, you could still launch a process that doesn't use capabilities.

PROFESSOR: No. Actually, no, you can't. You have to make sure that-- otherwise you could escape, like well, I can't access-- why don't you run this guy? [INAUDIBLE]. So yeah, cap_enter is inherited by all the children, which is actually hugely important. Yeah?

Anyone else? So what if we kill the UID? So it's supposed to be like going to cap_enter, and we just kill the UID of the current process. We don't actually care what it is anymore. And then the process tries to open a file. What checks should apply? Yeah?

AUDIENCE: Oh, I was thinking that the UID is useful for logging purposes as well, like being able to tell if you did something.

PROFESSOR: So yeah, you're right. Actually, yeah. So that would be actually kind of damaging, right? Like I spawned some sandbox process on my machine and it loses the UID. I'm like I have a hundred processes running on my machine, and I have no idea what they are.

So that's probably not a good plan for a management purpose. You're absolutely right. But I'm just sort of hypothetically saying, well, do we need it for access control, I guess. Yeah?

AUDIENCE: Maybe if this UID is only supposed to be able to access this file by reading or whatever, but you have the file descriptor for it, but then if you lose the UID, you might get permissions to write [INAUDIBLE] or something?

PROFESSOR: Yeah. I think actually what it shows up in is in directories. Because once you add a capability to a file, that's basically it. You have it open with particular privileges, et cetera. But the problem is that they have this hybrid design where they say, well, you can actually add capabilities to directories, and you can open a new file as you're running along. And it might be the case that you add a capability to a directory, like /etc. And you don't have access to necessarily all the files in /etc. But once you enter capability mode, you can now try to open those files by saying, well, I have access to the /etc directory. It's open already. Why don't you give me the file named password in that directory?

And the kernel still needs to make an access control decision on whether to allow you to open a file in that directory with either read mode or write mode or what have you. So I think this is the one place where you still need this ambient privilege, to some extent, because they're trying to build this compatible design where you can have semi-natural semantics for how directories work. Does that make sense? it's like one leftover place, kind of for compatibility reasons, or at least the way that Unix file systems are typically set up.

AUDIENCE: Are there any other places?

PROFESSOR: Good question. I couldn't think of one off hand, but I guess I would have to get their previous desource code to really figure out what's going on. I think most of the other situations don't really require a UID check. Because for networking, it doesn't show up. I think for process descriptors it doesn't show up, either. If you have it, then you just have it. So I think it probably is just file system operations.

For shared memory, it's also-- once you have a shared memory segment, you have it open. Yeah?

AUDIENCE: Could you explain again how exactly the user ID matters if you have a capability?

PROFESSOR: Yeah. So I think where it matters is, you have a capability to a directory. The question is, what does the capability represent? So one interpretation that-- for example, some capability system state, not Capsicum. Pure capability systems. They say, well, if you have a capability to a directory, then of course you have access to all the files in that directory, no questions about it.

And in Unix, this is typically not the case. You can open a directory like /etc, but there's lots of system files in there that are maybe private, like the private key of your server is stored in there. And just because you can look at a directory and open it and list it doesn't mean that you cannot open the files in that directory.

So in Capsicum, if you open a directory like /etc, and then you enter capability mode. And then you say, well, hey, I don't know what this directory is. I just add a file descriptor to it. There's a file in there called "key." Why don't you open that file "key"? And at this point, you probably don't want to allow this capability-based processor to just open it, because that wasn't the intent.

They'll allow you to bypass the Unix permissions on a file. So I think the authors of this paper are careful to design a system which would not violate existing security mechanisms.

AUDIENCE: So you're saying that you can, in some cases, use a combination of the two? So even though it'll be able to change it to directory, inside the directory, which files you can access depends on your user ID?

PROFESSOR: Yeah, exactly. So in Capsicum, the way they get it to work in practice is that, actually, before you enter capability mode, you have to guess. Well, what files am I going to need later? I'm going to need some shared libraries. I'll need some text files. I'll need some templates. I'll need some network connections, et cetera. So you open all these things ahead of time.

And you don't always necessarily know which exact file you need. So what these guys support as well, you can actually just open a directory file descriptor, as well. And then I can look up the particular files later. But it might be that the files don't have all the same permissions. So that's exactly the reason, yeah. Make sense? All right.

So this is the kernel mechanism part of it. Why do they also need this library for libcapsicum? I guess there's two things that they support in that library, as far as I can tell, or two main things. One is that they implement this function they call lch_start that you should use instead of cap_enter. And the other sort of feature the library provides in libcapsicum is this notion called fd lists instead of passing file descriptors by number.

So this fd list thing is probably the easiest thing to explain. It's basically a generalization, or maybe a clean up, of how Unix manages and passes file descriptors between process. So in traditional Unix and Linux, how you use it today, typically when you launch a process, you can pass it some file descriptors. You just open some file descriptors at particular integer numbers in this table and you run the child process that you want to run. Or you run a particular binary, and it inherits all these open slots in the fd table. But there's no real good way to name these things other than by number.

So the somewhat surprising convention, if you haven't [INAUDIBLE] before, is that, well, slot 0 is your input. Slot 1 is your output. Slot 2 is where you should print error messages to. And that's how Unix sort of works. And it sort of works OK if you are just passing these three files or streams to a process. But in Capsicum, what's happening is that you're passing down many more file descriptors around. So you're passing a file descriptor for some files. You're passing a file descriptor for a network connection, for a shared library, what have you. And it becomes much more tedious to manage all these numbers. So basically, libcapsicum provides an abstraction for naming these past file descriptors between processes by some sort of a hierarchical name, instead of just these opaque integers, if you will.

So that's one sort of simple thing that they provide in their library. So I can pass a file descriptor to a process and give it a name. And it doesn't really matter what number it has, a little easier. That make sense? OK.

So then they have this other mechanism, this much more elaborate way to start a sandbox. This lch, libcapsicum Host, API for starting a sandbox, instead of just entering the capability mode. So what happened? Why do they need something more than just entering capability mode? What are you worried about on creating a sandbox? Yeah?

AUDIENCE: It erases all the inherited stuff to give you a clean start.

PROFESSOR: Yeah. So I think they worry about trying to enumerate what are all the things the sandbox has access to. And the problem is that if you just call cap_enter, technically, at the kernel mechanism level, as we talked about just now, it worked. Right? It just prevents you from opening any new capabilities. But the problem is that there might be lots of existing stuff that the process already has access to.

So I guess the simplest example is maybe there are some file descriptors that you forgot you had opened, and it'll just get inherited by this process. So one example is they were looking at tcpdump. And they realized that-- well, first, they changed tcpdump just by calling cap_enter at the point just before they were about to parse all the network input.

So this works well, in some sense, because you can't get any more capabilities. But then they looked at the open file descriptor, and they realized that you have complete access to the user's terminal, because you have an open file descriptor to it. So you can actually sniff all the keystrokes that the user is typing and all that stuff. So it's probably not a great plan for tcpdump. This compromise you probably don't want sniffing everything you're typing.

So instead they-- well, in tcpdump's case, they manually changed these file descriptors to add some capability bits to them, to restrict what kinds of operations you can do. So remember, the capability, at least in Capsicum, has these extra bits that say, here's the class of operations you can perform on a file descriptor. So they basically take what used to be file descriptor 0. It pointed to the user's terminal, tty.

And originally, this was just a direct pointer to the tty structure in the kernel. What they do is they actually-- in order to limit the kind of operations you can perform on this file descriptor, they basically introduced some extra beta structure in the middle. This guy will point to the terminal. And the file descriptor itself will point to some sort of a capability structure. And inside of it is the pointer to the real file that you're trying to access, as well as some restricted bits or permissions on that file descriptor object that you can do.

In their case, they basically can say for tcpdumps standard input, you cannot do anything on it. You can just see that it exists, and that's it. For the output file descriptor, they say, well, you can write to it, but you maybe can't reposition. You can't [INAUDIBLE] back and forth, et cetera. Make sense?

So what else would you worry about, in terms of starting a sandbox? So there is, I guess, the file descriptor state. Anything else that matters? Well, I guess in Unix it's file descriptors and memory. That's pretty much it.

So the other thing that these guys worry about is that it might be that in your address space, you previously allocated some sensitive data. And the process that your sandbox is going to be able to read all its memory. So if there's maybe some password that you checked before when the user was logging in, and you haven't cleared that yet, well, the sandbox process will be able to read that and do something maybe interesting to that.

So the way they solved this problem is, in lch_start, you basically have to start a program fresh. You basically take a program. You explicitly package up all the arguments you want to give it. You explicitly package up all the file descriptors you want to give it. And then you start a new process, or you would call executives to reinitialize your whole virtual memory space.

And then there's no question about what is the set of sensitive data of extra privileges that this process has. It's exactly what you passed to lch_start, in terms of a program name, arguments, and capabilities. Does that make sense?

AUDIENCE: What would happen if the process that you're starting is a setuid 0 binary?

PROFESSOR: Yeah. I think these guys say that they don't actually allow setuid binaries in capability mode, just to avoid some weird interactions that would show up. I think the rules that they implement is that you could have a setuid program that gets its privileges from a setuid binary, and then it can call capenter or lch_start. But once you're in capability mode, you cannot regain extra privileges.

In principle, this could work, but it would be very weird. Because remember, the only place where the UID matters, once you're in capability mode, is in opening these files inside of a directory. So it's not clear this is really a great plan for getting more privileges or [INAUDIBLE] there. Make sense? Yeah?

AUDIENCE: We talked about earlier why the library doesn't really support strict separation between those two. And then we just mentioned all these problems that you could use [INAUDIBLE], so we're still not under a restriction to use lch_start necessarily, right?

PROFESSOR: That's right. Yeah. So lch_start, here's sort of the way to think of it. So you have an application, like maybe tcpdump. Or gzip is the other thing they work with. And what you're basically assuming is the application is probably not compromised, and there are some core part of the application that you worry about sandboxing. In tcpdump's case, it's actually parsing packets coming from the network.

In gzip's case, it's actually taking the file and decompressing it. And you're basically assuming, well, up until a point, the process is probably doing all the right things. It's not exploited. There's probably not a bug yet for the [INAUDIBLE] even. So at that point, you're trusting that it will run lch_start correctly and correctly set up the image, correctly set up all the capabilities, and then restrict itself from making any further system calls outside its capability mode.

And then you run the dangerous stuff. And by then, this setup has happened correctly, and there's no way to escape out of that sandbox. Make sense? All right. So I guess let's look at how you actually use capability mode to sandbox applications.

So we talked a little bit about tcpdump. How do you isolate this process? Another interesting example they had was this gzip program that compresses, decompresses files. So why do they worry about sandboxing it? I guess they worry that the decompression code is going to be potentially buggy, or maybe there's some memory management errors in how they manage the buffers during decompression, et cetera.

So could they-- well, one interesting question, I guess, is why are the changes to gzip seemingly much more complicated than for tcpdump? Any guesses? Well as far as you can tell, it's mostly just a question of how the application is structured internally, right?

So if you had a application that simply compressed a single file, or decompressed a single file, then it might be OK for us to just run it in capability mode without really changing it. You just give it a new standard in for something to decompress, and the standard out goes to the decompressed output, and that would work fine.

The problem, as is almost always the case here with these kind of sandboxing techniques, is that the application actually has much more complicated logic around it. So gzip, for example, can compress multiple files, et cetera. And in that case, you have some sort of a driver process on top which actually has these extra privileges to open multiple files, to create things, et cetera.

And the core logic needs to be often another helper process. And it was just so the case in gzip that the application wasn't structured in a way where this was already a separate process doing all the decompression or compression. So they had to change gzip's core implementation, and, well, some structure of the gzip application, instead of just passing the data to the decompression function to actually send it over an RPC call or really just write it to some almost file descriptor to help process the problems on the side and performs all the decompression with almost no privileges. The only thing it can do is return the decompressed data, or the compressed data, back to the caller process.

That roughly make sense? What's going on in gzip? All right. So I guess one thing we asked for the homework is how do you actually use Capsicum in OKWS? So what do you guys think? Would it be useful? Would the OKWS guys have been excited and switched to FreeBSD because this was much easier to use? Or is this a wash? So what do you think? How would you use Capsicum in FreeBSD? Would this be much different? Yeah.

AUDIENCE: So it means you can get rid of some of the jailing [INAUDIBLE].

PROFESSOR: Yeah. That's true. So truth seems to be completely superseded by this plan of having directory file descriptors and capabilities. So that's great. So you don't need the chroots setting it up. That seems messy. And this is much more precise, also. Because you can-- instead of having a chroot with lots of little things in there, you have to maybe set the permissions on there carefully. You can just open exactly the files that you need. So that seems like a plus. Any other benefits? Yeah.

AUDIENCE: [INAUDIBLE].

PROFESSOR: In OKWS, you mean?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. So in OKWS, right, you have this OK launcher daemon that had to launch all these guys. And it was the parent process. Only when they die, the signal goes back to this okld to restart the crash process. And that thing had to run this root, because it had to sandbox things. There's actually a number of things you could do better with Capsicum in OKWS.

So one example is you could probably have okld have many fewer privileges. Because it might need to be root initially to get fort 80. But after that, it could set up sandboxes for everyone else without being root anymore. So that's kind of cool. And maybe you can even delegate the job of responding a process to someone else, maybe a per service monitor Damion that just has this process descriptor handle, or process descriptor for child process, and whenever it crashes, starts a new one.

So I think this process [INAUDIBLE] helps things a lot. And the fact that you can create a sandbox without being root is also quite helpful, as well. Any other stuff, what you could do? Yeah?

AUDIENCE: You could give each one a file descriptor with append only mode to the log.

PROFESSOR: Yeah. So that's pretty cool. So as we were talking last time, in OKWS, well, the oklogd maybe could hamper with the log file. And who knows what the kernel will allow it to do once it has a file descriptor on the log file itself. But here, the fact that we can do much more of a precise capability map on a file descriptor, well, we could give it a log file and say, well, you could just write to it, but not seek. So that basically means append only, if you're the only writer to that file. So that seems kind of nice.

And you could prevent it from reading a file. You could say, well, you can only write, but not read, which is something that's probably difficult to do with Unix permissions alone right now. Make sense? Any other ideas for how Capsicum might help? Would you wish there was more stuff in Capsicum? I guess we always wish there was more stuff.

AUDIENCE: So one thing that perhaps may be tricky is the service team daemons need to connected to their backend databases somehow. Which might be remotely. But you don't want the launch daemon to know about which services each service is going to connect to.

PROFESSOR: Maybe, yeah. That's a good question, right? So in Capsicum, as we were talking about, the network is in global namespace. You have to have existing file descriptors for all the outstanding connections ahead of time.

AUDIENCE: Right. But you don't necessarily want okld to open up all the sockets for all the services. Because it might not know where the services are connected.

PROFESSOR: That's right. Yeah. So that's a little bit of an awkward thing. I absolutely agree. And this is part of the reason why I think capabilities haven't completely subsumed everything in the security world, is because they are kind of awkward to use. Because the guy that gives you all the privileges has to know exactly what things you're going to need, like these connections to backend servers.

So at some level, maybe this is not such a huge problem in OKWS. Because the launcher daemon has to read a Config file and is going to pass the token to the service in the first place. So maybe the token is going to contain the host and port number to which you're connected to. But I agree. It's not great.

Because especially, suppose the database server disconnects you. Well, you're kind of stuck now. The file server is not connected anymore, and you can't connect to a new one. So basically, if the database server crashes, or restarts, or the network breaks, you basically have to terminate it, get yourself response, so you can get a new one of these connections past you. So it's maybe not a great plan in that sense.

AUDIENCE: Could we wrap the system call, the function [INAUDIBLE] to open a socket so that it faults the middleman instead of the socket that the users send out to [INAUDIBLE]?

PROFESSOR: Yeah. This is what I think the FreeBSD guys have done since. Well, there's a bunch of situations like this, where you want to open some file after the fact, or you want to connect to something after going into capability mode. So the FreeBSD developers have added this daemon called Casper, that every capability based process has a handle on. And this Casper daemon runs outside of capability mode, and basically listens to requests from sandbox processes.

And if you want to open some file, or if you want to send a network connection, or a packet, or something, but you didn't have the right capability beforehand, then this Casper daemon will do it for you. But it carefully maintains a list of things that every sandbox process should or should not be able to do. So it's like a systems service. So when you start a capability process, or enter capability mode, by default, this Casper thing will not allow you to do anything extra funny.

But you could say, well, hey, I'm going to start the sandbox process. And you can ask Casper, well, please allow my process to do the following things later. So you could, right? And the cool thing is that you can pass file descriptors or capabilities through fd passing in Unix. So once you have a handle on this Casper guy, you can get more capabilities later on. So it's, again, trade off between being pure capability world versus actually being programmable or easy to use.

So it seems to be working out. I think the particular thing they use it for in FreeBSD, or the thing that shows up often, is making DNS queries. So you want to be able to make DNS queries once you're in a sandbox. And actually, this is a problem they ran into with tcpdump. Because when tcpdump is printing your packets, it wants to print the host name for an IP address.

In order to do this, it has to talk to a DNS server. But you probably don't want to connect to a DNS server ahead of time, or to every DNS server you might ever need. So instead, they use this helper daemon that's going to make DNS queries for you. Make sense? All right.

So I guess the last thing I wanted to talk about is what are the security guarantees that Capsicum provides? So should you trust it? How could Capsicum go wrong? Presumably you can always have security problems, regardless of what mechanism you're using underneath. But what particular things should we worry about in Capsicum when we're building some system here?

Suppose you have to attack this thing. You have to attack this tcpdump thing, or gzip, or whatever it is that they implemented. What would you look at, in terms of bugs or problems?

AUDIENCE: Well, it depends on the developers knowing what they're doing. So they might give a bad capability.

PROFESSOR: That's right. Yeah. So it's actually one interesting property of Capsicum is that it's not a guarantee that the user of the system gets. It's really a tool that the developer has to build more trustworthy or better application software. But I, as a user of the system, have no idea whether this is a good or bad thing that the application is using Capsicum. You could totally misuse it, as you're absolutely right.

So maybe one example is, as they show in the paper, you could give too many privileges to the sandbox process. Like the the TCP helper, or maybe it has access to my console. And that's not so great, but it's hard for me as a user to really tell this in a general purpose fashion. Yeah?

AUDIENCE: It might also be that when you set the permissions to the masks on any given file descriptor that you set two permission masks.

PROFESSOR: Yeah. Right. So it's not just the file descriptors. Also, what can you do with those file descriptors? You're right. Yes. These maps are another part of the story that you have to watch out for. OK. So suppose we got the masks right. We got the file descriptors right. We haven't used lth_start. There's nothing extra in memory.

AUDIENCE: [INAUDIBLE].

PROFESSOR: That's true. Yes. So maybe there's like something before you even add the capability mode that's damaging. So it only helps once you jump in. And one slightly annoying thing is that it seems like it can't do a whole lot inside of capability mode, not in the sense that you can't run large computations, but you can't really put a large part of a complicated system into capability mode. Because inevitably, in Unix, you'll need to do something with new processes, opening network connections, et cetera.

And you'll probably need to use some of these global namespaces that are not available in capability mode. So it's probably going to be quite difficult to put large chunks of logic or intricate system code inside of capability mode. So only well-defined chunks of an application are likely to be running in capability mode. It depends. I don't know if this is entirely true or not. In Chrome, for example, large processes do run in capability mode in their design.

It might be that you basically have to have non-capability mode chunks of your application because you wanted to incorporate nicely with Unix, or whatever is is you're running alongside of it.

OK. Any other thing you should worry about? Yeah?

AUDIENCE: Well, whether they implemented capabilities correctly.

PROFESSOR: Yeah.

AUDIENCE: Whether they've covered all the system calls.

PROFESSOR: That's right. Yes. So that's actually a huge problem, in some sense, already. If you think about it, there's probably hundreds of system calls that the kernel provides you. And they're not especially precisely documented, so you probably have to look at their implementation and see if, for every system call, if there's some way for the applications to get the system call to perform some operation on some extra object that didn't have a file descriptor to it.

And most Unix system calls weren't written with the expectation of everything has to be operation on a file descriptor. So you really have to get every system all right. And probably more worryingly is that the kernel has to be free of bugs, like buffer overflows or whatever other memory corruption like you guys explained [INAUDIBLE].

Otherwise, all of this is complete nonsense. You just are on arbitrary assembly code in the kernel, and you have full control of the machine.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. I guess, yeah. So the one thing I didn't get a chance to talk about is alternative things. So this is in FreeBSD. Linux has this thing called [INAUDIBLE], that allows you to specify which system calls you can operate. If you squinted, it's kind of like Capsicum but very different, in the sense that Capsicum talks about specific file descriptors that you can operate. And in Linux, the [INAUDIBLE] mechanism lets you talk about specific system calls that you could run.

So it's probably less fine grained, but it's what's available in Linux today. And it's actually probably a good idea to look at your applications and see what system call do you expect it to make and then code in a filter and allow it to make only those system calls. The problem is that if you have any interesting applications, it'll probably run exec and open and write, and that's probably enough to do quite a bit of damage to the system. So that's why you probably want the more fine-grained system like Capsicum, where you can say, well, you can run right, but only on this thing, not on my entire home directory.

All right. So I guess we're out of time to talk about Capsicum. So let's talk about native clients on Wednesday and a different way to sandbox programs.