PROFESSOR: Today we're going to talk about privilege separation, so we're done with buffer overflows at some level, but will keep coming back as being a problem that we want to deal with, so we'll not talk about the details of how to exploit them, now we'll switch more into mitigation, if you will, or prevention techniques of how do you design a system where buffer overflows aren't such a huge problem for you, perhaps, as well as other security vulnerabilities.

So for today we're going to talk about privilege separation as a general purpose technique for how to build a more secure system, and the particular paper we assigned for you today is this web server called OKWS. It's not necessarily the biggest example of privilege separation on there, but it's a reasonably well described system that we can actually read and really understand how all the pieces work, and you should really think of it more as a case study of how to do privilege separation right. Not necessarily you should go and download OKWS to run your website right now.

So before we dive into the details of OKWS and Unix permissions, let's just see what is privilege separation, why is it such a good idea? And then, last week's lecture, James showed you that if you write a program in C, then it's almost inevitable you'll have something bad go wrong in that program, and the problem, at some level is that if you have a large application and there's any kind of all vulnerability in this application, then adversaries can connect and send requests for this application, might be able to trick it into doing bad things. And the application is presumably privileged, meaning there is probably lots of data sitting behind the application that it can access and maybe delete files, like you guys are going in Lab I now, read sensitive data, install back doors.

And the problem is that a vulnerability in this large application can allow it to modify any of this data, or basically exercise all of the privileges this application has, and it probably has lots of privileges, unless you're careful about it. And what privilege separation tries to do, and what we'll look at in this lecture, is to take the application and chop it up into different pieces and make sure that each piece has only the necessary privileges to do its job correctly.

So you could imagine maybe all the privileges you care about are access to data in the back end, then all of this data, maybe you can slice it up in some way, give this access to this piece of data, this piece access to this piece of data, and so on. So then, if you find a bug here, then maybe this data is kind of compromised, but hopefully whatever slicing you've done is going to enforce the separation so that a vulnerability here doesn't allow the attacker to go and access these other pieces of data, or, more generally, other privileges of the application has access to.

So this is the big idea behind privilege separation, and it's hugely powerful. It actually doesn't really rely on buffer overflows or other kinds of vulnerabilities being present. It's just a general architecture for making sure that vulnerabilities in one place don't affect as much as possible your system.

This turns out to be used pretty widely. Virtual machines often are used for enforcing isolation within components. Maybe you'll take your large system and break it up into a bunch of DMs for isolation, but you could also use Unix to actually perform this isolation with slicing. And as we'll talk about in a second, Unix does provide you quite a number of mechanisms that OKWS does actually use to achieve privilege separation.

And then many applications actually use privilege separation practice. You guys are probably using SSH quite often. That uses privilege separation in many of its components to make sure its keys are not leaked and the server doesn't get compromised or the effect of our server compromise is reduced. And perhaps more relevant to you guys, Chrome, the web browser, actually does privilege separation quite extensively as well. So that if there's a bug in Chrome's implementation, the adversary doesn't get full control of your computer, which is a great property to have.

So that's just a very quick summary of what privilege separation is about and why maybe OKWS is an interesting case study. I guess we can add it to this list, but it is more of an illustrative example rather than an important piece of software in its own right. Make sense? Any questions before we dive in?

All right. So OKWS, as I mentioned, it's going to use Unix permissions and sort of Unix mechanisms to achieve the separation between its different components. So as a result, it's going to be important for us to understand how Unix protection mechanisms work. And Unix isn't in some way crucial to OKWS at some level for privilege separation, but for any isolation mechanism you're going to use, whether it's Unix, uid, these other mechanisms, or virtual machines or containers or any other technology, it's really important to understand the details of how the isolation mechanism works, because there's a lot of tricky pieces to get right, because you're dealing with some attacker that can exploit any [INAUDIBLE].

So as a result, we'll look at Unix in a fair amount of detail just to see what it's like, how should we approach thinking about a particular security mechanism. Let's look at Unix. So Unix historically-- well, it's not necessarily the best example of how to build a security mechanism, because its security mechanism came about from a fairly utilitarian need of needing to separate different users on a single Unix system from one another, so they weren't thinking of it as a general purpose mechanism that applications like OKWS are going to use to implement privilege separation.

They're just thinking, we have a bunch of users that are using the same computer, we need to keep them from each other. So it's not necessarily a general purpose mechanism but still one that is fairly prevalent and, as a result, widely used. Chrome tries to use many if these Unix mechanisms.

So what does Unix have? So, in general, when you're thinking about protection mechanism, you should be thinking, well, what are the principals, meaning what are the entities that have privileges or rights, and in Unix these principals are typically invoked, or sort of held, by a process. So I guess the subject, if you will, in Unix is a process, so every operation or request that we can think about in terms of security, whether something should be allowed or not, is probably going to be an operation that a process invokes by making a system call.

And the principal is how we describe what privileges that process has. Conversely, there's also what we can think about as objects, and these are the things that a process might act on that try to modify, read, observe in some way. There are actually a lot of different kinds of objects you might worry about protecting in an operating system. What do you guys think? What should we worry about protecting?

AUDIENCE: Files.

PROFESSOR: Files. Yeah, great. That's a big one. That's where all of our data lives, right? There's a closely related thing we might worry about-- directories. Turns out to be pretty important also from a security standpoint. Anything else?

AUDIENCE: Networking sockets.

PROFESSOR: Yeah, great. Networking sockets. Anything else going on?

AUDIENCE: Other processes.

PROFESSOR: Oh, yeah. Actually, this is like stuff that the application or the user might care about, but then there's all kinds of internal stuff that you have to protect as well, so a process is not just the subject that's making a system call, but a process is also something that another process can act upon. It can kill it or create a new one. You have to figure out, what are the rules for thinking about process as an object you can manipulate. Other things we might care about?

AUDIENCE: Environment variables.

PROFESSOR: I guess they're probably not an entity you can modify, in the sense of being managed by an iOS and having some sort of a security policy. I guess I sort of think of environment variables as just being some state a process maintains in memory. But, I guess more generally, we do care about maybe part of a process is all this stuff in memory. So there's going to be environment variables there, there's a stack, there's arguments, and this also turns out to be quite important. Presumably lots of sensitive data lives in processor's memory. Other things?

AUDIENCE: File descriptors in general.

PROFESSOR: There's like another sort of internal detail that matters a lot. So files are the stuff we might care about on disk, and there's this operational thing, the file descriptor, that OKWS makes quite extensive use of, and we'll see what file descriptors are in a little bit. Any other stuff you guys want to protect in an operating system?

AUDIENCE: Hardware.

PROFESSOR: Hardware? Yeah, I guess in many ways hardware is-- well, hardware is, in some ways, not really an abstraction that the iOS provides to you. I guess you run a process, so you might want to make sure the CPU doesn't get stuck.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Oh, yeah, yeah. So, like, extra devices, Yeah, you're right, especially on a desktop machine, there's lots of extra stuff. So there's your USB drive you plug in, your webcam, probably your display itself is something you'll want to protect, like an application shouldn't draw all over your screen anywhere. So, yes, actually I guess this isn't really on a server side view of things, where there's just a server somewhere in a closet, but on your phone, your microphone probably, is a hugely important object that you want to protect, yeah, but I will also leave it off this list, because we're going to talk much more about server applications for now, but you're absolutely right.

I think for OKWS, this is probably a more or less exhaustive list of things we might care about protecting, or, at least that OKWS uses. So let's talk about how does the OS kernel decide when a process can do something to any of these objects? So the [INAUDIBLE], I guess is, we mostly think of a process as having the privileges represented by this principal, and the principal in a Unix system is this slightly complicated thing. There is something called a userid, which is just a 32-bit integer. There's also a group ID, which is also a 32-bit integer. And there's not really a great reason why they're different. It would've been nice if they were just a uniform set of 32-bit integer principal numbers, but unfortunately Unix sort of splits them into two categories. There's userid integers and then there are group ID integers. When we talk about a process having certain privileges, we typically think of a process being associated with a particular uid value. The process, for the most part, has a single uid. As with almost everything else, there's complications everywhere in Unix, but I'll simplify it for now. A process has one uid, and there's also a list of group IDs that a process has. For historical reasons, the group IDs are split into one and then a list of others. Roughly, a process can then exercise the privileges represented by all of these identifiers. So if there's something accessible to this userid a process can do stuff with it.

That's how we think about what privileges a process has, so now let's talk about files, directories and other kinds of objects. So what happens with files, or how do Unix permissions for files work? Well, in Unix, every file has-- actually, maybe a better way to start is to think of what operations do we care about? For files, things are relatively straightforward. For files, you probably care about read, write, maybe things like execute as well, change permissions, maybe change other security properties.

AUDIENCE: Unlink.

PROFESSOR: Unlink. Well, so is unlink a property of a file itself or is it a directory thing? Actually a little not clear. At least, the way Unix thinks of deleting a file, is that it's really a directory kind of thing, because in Unix you can have-- a file is really an inode, and in Unix you could have multiple hard links to an inode and when you unlink a particular name of a Unix file, what you're really doing is killing one of the names for that file, but it might have other names, other links to it. So what actually matters is whether you are allowed to modify the directory pointing at the file and not do something to the file's inode itself. So typically, unlink and link and rename, create, are operations that we sort of think of as being associated with the directory, although, they are actually related, so "create" affects both the directory and a new file as well, so we have to figure out what are the rules there.

OK, so what are the rules? In order to help us decide when someone can read or write a file, we're going to stick some permission stuff, or bits, in the file inode. In Unix, every inode, meaning something that ends up being the file or directory, has a couple of interesting fields for security purposes. There's a userid and a group that we say owns the file or owns the directory. So you might have all the files in your home directory are probably owned by your on your Unix system.

There's also a set of permission bits in Unix that you can sort of think of as a bit of a matrix, so we want to have-- well in Unix there's basically the basic design, there's read, write and x for execute permissions. We can specify these permissions for different entities, and in Unix these are specified for the owner, meaning for the uid of the inode, for the group that owns the file, this gid and everyone else, other. You can sort of fill in this 3 by 3 binary matrix. You might say, well, I can read and write and maybe not execute this file. People in that gid might be able to read but not write this file, and everyone else-- or maybe they can also read it-- but not do anything else with it.

So this is the way Unix stores permissions. There's some baroque way of encoding these things that you'll see often that's probably worth mentioning. In Unix, you encode this matrix as an octal number, so you treat each row here as a base 8 number, so r is bit 4, w is bit 2, x is bit 1, so this ends up being 6, 4, 4, so you'll sort of say-- well, you'll often see this notation, even in this paper. You'll say, well, this file has permission 6, 4, 4, meaning the owner can read and write this file, the group owner can read it and everyone else can also read it. Does that make sense?

This tells us when you can read, write and execute a file. What about changing permissions on a file? This is not entirely a fair question, but what do you guys think? How should we decide when someone should be able to change these permissions, because that's also something to try to do, at least.

Any guesses? Yeah.

AUDIENCE: If they have [INAUDIBLE].

PROFESSOR: Maybe, yeah. It depends. On the other hand, you might create a overwritable file that I just want to share with anyone, that you can read and write and modify my file, but then this also means that you'll all of a sudden be able to change permissions, so you'll be able to take my file and make it not overwritable or take it over. That seems not necessarily great, so in Unix, what are the designers chose, is that, well, if you own the file, meaning if you have the same uid as the file, then you can change permissions. Otherwise, you cannot. So even if you're in the gid here and that group has all the permissions in the file, you still cannot really change the permissions on that file. You can just read, write, execute, whatever to get that solved. Make sense?

Then directories actually in Unix follow a pretty similar story. So unlinking and linking entries in a directory means having write permission on that directory, and if you want to rename a file, then you probably need to have write permissions on both the directory you're moving it from and the directory you're moving it to. A fairly natural plan. There are some corner cases with hard links, as it turns out. Lecture notes have some details but, more or less, that's how it works.

There's actually another interesting operation on directory that you might care about, which is lookup. So you might want to just look up a file in a directory. And Unix sort of encodes execute permissions as implementing lookup for directories, so what it means to have execute permissions on a directory is just being able to look up a certain name there. Might be that you don't actually have to execute permission on a directory so you can look up a name, but you don't don't have read permission, so you can't list the contents of a directory. It turns out to be useful in some situations if you really want to restrict what someone could do with those files, or sort of hide the files from a user.

Let's just work through an example. What happens on Unix if I call open("/etc/password")? What checks is the kernel going to perform on my behalf when I issue this system call?

AUDIENCE: It checks whether you have execute permissions on etc?

PROFESSOR: Yeah, that will happen somewhere. I need to execute on etc.

AUDIENCE: And then execute on slash.

PROFESSOR: Yes, actually, I need to look up what does /etc even point to? So if I don't have look up permissions on root, then that's not going to work.

AUDIENCE: Then you need read on /etc/password.

PROFESSOR: Make sense, roughly? Here's a small puzzle. Suppose that MIT sets up a group for all the people associated with 6.858 and another group in the Unix sets of gids for all the TAs at MIT, but they don't have a group four 6.858 TAs for some silly reason. Could I create a file that's only accessible to 6.858 TAs? If I have a 6.858 group, or some gid, and a TAs gid. So there's only one gid that I can stick in a file. Any guesses?

AUDIENCE: Well, you couldn't anyway because you might have TAs that and not 858 TAs.

PROFESSOR: That's true, yeah. Suppose they want to-- you're right, yeah, so there are students in 858 that are TAs of other classes, so that's maybe not great. But, still, lets try to do intersections somehow.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yes, so you can actually play tricks with this mechanism. It's not perfect, but you can sort of use it to encode more interesting things. You could actually do something like create /foo/bar/grades file, and what I'll do is I'll actually make foo owned by-- or set the gid to 6.858 and only make it executable for the group.

So unless you're in this group, you can't even look things up in /foo, and then I could set the permissions on bar so the gid is for TAs and then it's executable for group as well and not others. So unless you can actually traverse this path, you can't get to this grades file. It's kind of, you know, a cute hack, if you will, but these kinds of tricks are things you end up doing with whatever the base primitives are, the isolation mechanism provides to you. And even OKWS plays all these tricks as well in their design. Make sense? Questions? Yeah.

AUDIENCE: If, like, the permissions on the grades file itself, were let's say-- the QID was 6.858, could a TA, like, link it to some other directory and allow anybody in 6.858 to access it?

PROFESSOR: Yeah, potentially, right? So you might worry about other things like leakage now, so Unix in general doesn't try to enforce these kinds of transitive security properties, meaning that once a process has access to some data, or has some privileges, it can basically delegate those privileges to anyone it wants. There are other kinds of systems called mandatory access control systems.

We'll perhaps talk about them later, but those try to actually enforce this transitive property that, if I give it to you, then you can't give to other people. You're basically stuck. It sort of taints you and you can't go anywhere else. In Unix, this is generally not the case, and a TA probably could not hard link this file because of another silly rule that Unix enforces for hard links, which is that only the owner of a file can hard link it somewhere else.

And this is partly because of the way Unix does quotas, because in Unix quotas are by who owns the file. So if you create some giant file, I can hard link a copy over to my directory, then you maybe delete the file, but I still have it, and the file system thinks, yep, that's the owner, but you can't even delete it, because I have the reference to it. So that would be a bit of an unfortunate combination of Unix mechanisms there.

But in general, you should worry about such things like transitivity, like could someone-- or maybe a better problem is, maybe someone was a TA and then we remove him. But maybe they can still sort of stash away a reference somewhere, so this is maybe not a perfect solution for this problem for many reasons, including the fact that there's non-858 TAs taking 858. Are there questions?

OK, so that's files and directories in Unix, so how security works for them. A closely related thing in Unix are file descriptors. The file descriptors are used fairly pretty widely in OKWS and what a file descriptor represents in Unix is basically an open file. So in Unix in particular, it turns out that the security checks on opening a file are performed-- or security checks for accessing a file-- are performed when you open the file in the first place.

And from there on, you have basically a handle on the file, where anyone with that handle can now perform operations on that file. So the rules for basically accessing a file descriptor are, if you have an open file descriptor in your process, then you can access it. And security checks don't apply in the sense that, to get that file descriptor, you could have just opened the file, in which case these regular checks would have applied, or some other process might have passed the file descriptor to you, so you can pass file descriptors by inheriting from a parent, so a parent can pass a file descriptor to a child process or you can pass file descriptors through sockets in Unix, but however you manage to get a file descriptor, you can read and write the file descriptor all you want, because the security checks have already been done when the file descriptor was initially created.

So that's actually a nice way in Unix to give someone privileges that they don't otherwise have. So in OKWS there's probably many components that need to act as a certain socket or file, or whatever you have it, and one way to implement this without giving them direct access to read and write the file in the file system, is to have someone else open the file, create a file descriptor and then pass it to this extra component. This way, you can really precisely say, that's the only file descriptor you'll ever have. And there's nothing else they can try to do in the file system that might be funny. Make sense? So in fact, it has fairly simple rules, I guess. If you have a file descriptor, you can do whatever you want with It.

OK, so what about processes? What are the rules there? I guess, what can you do to a process? In Unix it's fairly simple. You could, I guess, create a process. You could kill it. You could debug it. There's this mechanism called ptrace in Unix, and probably a couple of other things. And the rules are relatively straightforward. So you can always create a process, more or less, except that the child process is going to get the same userid as you, so you can't create a process with some other userid by default. So you can't say, well, I'd like to create a process running as "web," one of my TAs. The operating system kernel will not let you do that. If you want to kill a process, you basically have to have the same userid as that process as well. It's kind of nice. All the things with a single userid are isolated from things with other userids. And more or less, the same rule applies to ptrace as well. The process with the same uid can debug processes with the same uid.

As with everything, it turns out race conditions show up often and can cause problems, but there have been actually some interesting bugs in the ptrace mechanism in Linux where, if you debug a process and then it switches and gets more privileges, then maybe you could somehow trick the kernel into letting you retain this debug privilege on this process, even after it becomes more privileged. Then you can monkey with its memory and take it over. But at least the basic design that you probably want to enforce is roughly a process with the same uid can act on each other, otherwise not.

And I guess, OK, so what else did we have on this list? Processes. Memory sort of goes along with the process. So, unless you're in that process you can't access the process memory. Virtual memory nicely enforces this isolation for us. Except this debug mechanism lets you poke in another process's memory if you happen to have the same userid.

And then, I guess the other remaining thing for us is networking, and networking in Unix doesn't really fall in the same model, partly because of it came about later. You know, the Unix operating system was designed first and then networking came along and became popular. It has a slightly different set of rules. So I guess the operations we really care about on the network is, presumably, connecting somewhere or maybe listening actually for connections as well.

So you might want to connect to some web server or you might want to run a web server yourself and listen on a particular port. Maybe you want to actually read data from a connection, or read/write data on some existing connection, or you want to send the raw packets or receive.

So in Unix the network stuff basically has no relation to userids, the first approximation. The rule is anyone can always connect to any machine or any IP address, can always open a connection. If you want to listen on a port, that's where one difference shows up, which is that most users are prohibited from listening on ports below a magic value of 1024. Basically, if you listen and the port is less than 1024, then you have to be a special user called "super user" with a uid of 0.

And in general, Unix has this notion of an administrator, or super user, which is represented by having uid of 0, which can bypass pretty much all these checks, so if you're running as root, then none of this applies. You can read/write files, you can change permissions on anyone's files and the operating system will let you do that because it thinks you should have all the privileges. And one thing you really need it for is for listening on ports below 1024. Any idea why this weird restriction? Who cares about your port number?

AUDIENCE: Would they define specific port numbers to be certain things, like HTTP is like 80.

PROFESSOR: Yeah, it's like HTTP is 80 here. On the other hand, other services might be above 1024, so why this restriction? Why is this useful? Seems to complicate my life more, after you.

AUDIENCE: Since you don't want random searches just listening on your HTTP.

PROFESSOR: Yes. I think the reason for this is that it used to be the case, at least, that you'd have these machines where there's lots of things running, there's users logging in, there's services running, and you want to make sure that some random user logging into our machine doesn't all of sudden take over the web server running on that machine, because people connecting from outside don't really know who is running on that port. They just connect to port 80.

And if I want to log into that machine and start my own web server, then I would just take over all the web server traffic to that machine. That is probably not a great plan. So this is one way that the networking subsystem in Unix prevents arbitrary users from impersonating what are called well-known services running on these low port numbers. So that's sort of one rationale for this restriction.

And then, in terms of reading and writing data on a connection, well, if you have a file descriptor for a particular socket, then Unix lets you read and write any data you want on that TCP or uTP connection. And then for sending raw packets, Unix is actually pretty paranoid about this, so it actually will not let you send arbitrary packets over the network. It has to be in the context of a particular connection, except if you're root, of course, then you can do whatever you want. That make sense? Somewhat? Any other questions about all this Unix machinery?

OK, so one interesting question we could try to ask, is where do these userids come from? So, when we talk about processes having a userid or having a groupid, and if you run PS on your machine, you probably see lots of processes with different uid values. Where do these guys come from? We need some sort of a mechanism really to bootstrap all of these userid values, and the way it works in Unix, at least at the mechanism level, is that there's several system calls for doing this.

So initially to bootstrap these uid values, there's a system called setuid() that you that you can pass some sort of a uid number to, and it'll set the userid of the current process to this value. This is actually a dangerous operation, of course, so in sort of Unix tradition, you can only do this if you're uid is equal to 0.

Well, must have. So if you are this root user with uid 0, then you can call setuid() and switch your user to anything else. There's a couple other similar system calls for initializing the gids associated with the process. It's setgid and setgroups.

So these system calls together let you configure the privileges that a process has. So typically, when you go and log into a Unix machine, the way that your processes get the right privileges, is that you're initially actually not talking to a process running as your uid, partly because the system doesn't know who you are yet. Instead, what you initially talk to in Unix is some sort of a login process, so maybe SSH runs a process for anyone that connects to it and tries to authenticate the user.

So this login process runs with uid=0 as root and then when the supply username and password, it's actually going to check it against its own database of accounts and, typically in Unix, this get stored in two files, /etc/password, which, for historical reasons, no longer stores the password. And there's another file, /etc/shadow, which does store the password, but in /etc/password, there is actually a table mapping every username in the system to these integer values.

So your username gets mapped to a particular integer number in this /etc/password file, and then login will check whether your password is correct, according to this file, and if it is, it'll find your integer uid and then call setuid on your uid value and then execute your shell. Whatever, (den/sh) And now you can actually interact with the shell, but it's running as your uid so you cannot do any arbitrary damage to this machine. Question?

AUDIENCE: Is it possible to start a new process with uid 0 if you have some non-0 uid? For example, if you want [INAUDIBLE].

PROFESSOR: Yeah, so this sort of lets you go down, if you will, so with your root, you can restrict yourself down to a different uid, but the rule we set so far is you can only create a process with the same uid as yourself. But, of course, you want to elevate your privileges for various reasons. You want to, I don't know, install a package now and you need root privileges.

So, Unix has basically two ways you could think about doing this. One way we already mentioned, this file descriptor passing thing. So if you really want to elevate your privileges, maybe you can talk to some helper, and the helper is running as root. You can ask it, hey, can you open this file for me? And maybe you like define some new interface, and that helper opens the file and gives you back the file descriptor through fd passing. That's one way you could elevate your privileges, but it's kind of awkward, because what you really want in some cases is a process running with more privileges. So in order to do this, Unix has this sort of clever, sort of problematic mechanism called setuid binaries.

So setuid binaries are just regular executables in a Unix file system, except that when you run them, when you sort of co-exec on a setuid binary-- one example is, for example, is /bin/su on most machines, or sudo as well. There's a bunch of setuid binaries on a typical Unix system. The difference is that when you execute one of these binaries, it actually switches the userid of the process to the owner of this binary.

It's a little bit of a weird mechanism when you first see it. Typically the way it is used is that this binary probably has an owner uid of 0, because you really want to regain lots of privileges-- you want to regain root privileges-- so you can run this su command, and the kernel, when you exec this binary, will switch the uid of the process to 0, so this program will now do some privileged stuff. That make sense?

AUDIENCE: If you have uid 0 and you change the uid of all of those setuid binaries to something non-0 and then you could start [INAUDIBLE].

PROFESSOR: Well, many processes will not be able to regain privileges later. You might be kind of stuck. It'll still boot probably, but maybe some things will not work. This mechanism is not tied to uid 0. In fact, "I" as a user on a Unix system can create any binary. I can build some program, compile it, and I can set this setuid bit on that program itself. It's owned by me, the user, my userid. And what this means is, anyone executing my program will run that code with my userid. Is that problematic? Should I do this?

AUDIENCE: So, if there was a bug in your application and suddenly someone could do anything as you, not just with the program that's assigned to you.

PROFESSOR: Right. But yes, so that's right. If my application is buggy, or if it allows you to run anything you want, well, I could copy the system shell and make its setuid to me, then anyone can run a shell under my account. That would probably not be a best plan of action. But a system mechanism, well, this is not necessarily problematic, because the only person that can set the setuid bit on a binary is the owner of the file, and owner of the file has that uid privilege, so I can basically give away my account to other people if I want, but someone else cannot create a setuid binary with my userid. That make sense?

And the setuid bit is sort of stored alongside these permission bits. So somewhere there is also a setuid bit in every inode that says whether this executable or this program should be switched to the owner's uid on execution. Does that make sense as sort of a privilege estimation mechanism?

It turns out that this is a very tricky mechanism to use correctly. So the kernel implements it correctly. It's actually a fairly easy thing to do. It's just one check. If there is [INAUDIBLE], switch the uid. Easy enough.

But using it safely turns out to be very tricky because, as was just pointed out, if this program has bugs in it or does something unexpected, then you might be able to do arbitrary things uid 0 or whatever the other uid is. And it turns out in Unix, the way you execute a program, you inherit a lot of stuff from your parent process.

For example, you can pass environment variables to the setuid binaries, and it used to be the case that-- well, in Unix, you can specify what shared library should be used for a process by setting an environment variable, and it used to be that the setuid binaries weren't careful about filtering out these environment variables, so you could run bin/su, but say, well, use my shared library for things like printf(), so your printf() is going to run when bin/su prints something out, and you can get it to run a shell instead of printing stuff.

So there's many other subtle things that you have to get right in terms of this program not trusting the user input, and this is actually quite different from how you think of writing most Unix programs. You generally do trust the user input a lot, so for this reason, the setuid mechanism hasn't been the most secure part, in some sense, of the overall Unix system. All right. Any questions about this stuff? Yeah.

AUDIENCE: Does setuid apply to groups as well, or just the user?

PROFESSOR: There is actually a symmetric setgid bit you could set. Why not. And you could--

well, the same thing happens, right? If the file has a particular gid and that setgid bit is set when you run the program, you get that group. It's not used a lot, but it is useful in cases where you want to give very specific privileges. So here, like bin/su probably needs a lot of privileges, but it might be that there's some program that needs a little bit of extra privileges, like maybe to write something to a special log file. So you probably want to give it some group and make that log file writable by that group. So even if the program is buggy, which is likely the case, then, well, you lose that group, sort of, privileges but not much else. It is sort of useful as a mechanism, but it doesn't show up often, because it's-- I don't why. People should use root more. Yeah.

AUDIENCE: What are the restrictions on who can change the [INAUDIBLE]?

AUDIENCE: Yes. Different Unix implementations have slightly different checks for this. The general rule of thumb is, only root can change the owner of a file, because you don't want to create files owned by someone else, and you don't want to take over other people's files, of course, either. So, in general, if you're a particular non-0 uid, then you're stuck. You can't change owner of any file. If you're a root, you can change it to anything you want. There are some complications if you're a setuid binary and you switch from one uid to another-- it's a little bit tricky-- but for the most part you basically can't change the owner of a file unless you're a root. Make sense? Other questions about this machinery?

It is, admittedly, a slightly baroque system. You could probably imagine lots of ways in which you could simplify this but, in fact, most successful systems sort of look like this as they evolve over time. As it turns out, you can make some good use of these sandboxing mechanisms. These are just sort of the basic Unix primitives that show up in pretty much every Unix-like operating systems, so Mac OS X has this, Linux has this, FreeBSD has this, Solaris-- if anyone's still runs this, et cetera. But in every one of these, there is actually more sophisticated mechanisms that you might use, so Linux has something called set COMP for sandboxing processes, Mac OS X has its own thing called Seatbelt, and there's all kinds of extensions. We'll at one extension actually next week, just to see, but this is just to get you familiar with the basics that every Unix system has.

So one sort of last bit of machinery we want to look at before diving into OKWS, is how do you deal with setuid binaries? How do you protect yourself from these security holes, if you will. So the problem is that inevitably you'll have some setuid binaries in your system like /bin/su, or sudo, or what have you, and there's probably bugs in these programs, so if someone can execute the setuid binary, then that process might get root access, so you don't want to do that-- or don't want to allow that.

The mechanism in Unix that is often used to prevent a potentially malicious process from exploiting setuid binaries is to use the file system namespace to modify it, using the chroot system call. OKWS uses this pretty extensively. So in Unix, what you can do is you can call chroot on a particular directory. So maybe you can chroot("/foo") and there's actually two explanations I want to give to what chroot does. The first one is just intuitive. What it does is it means that after you run chroot, the root directory or slash basically is equal to what /foo used to be before you called chroot. So it kind restricts your namespace down /foo so it looks like that's all the stuff you have. So if you have a file that used to be called /foo/x, after calling chroot, you can get at that file by just opening /x. So just restrict your namespace down to a subdirectory. So this is the intuitive version.

Of course, in security, what matters is not the intuitive version, but what is the kernel exactly doing with this system call? What the kernel does is basically two things. So when you call chroot a particular directory, it does two things. One, it changes what slash means, so whenever you access-- whenever you start a path name with slash, the kernel will now plug in whatever the file you gave to chroot. It's roughly the /foo file from before you called chroot.

The other thing the kernel does, is it tries to prevent you from escaping out of your / by doing /../ because you could imagine in Unix, I could ask for, you know, give me /../etcpassword. So if I just prepended /foo, then this would not be good, because I can just sort of walk out of /foo and go get /etc/password.

So the other thing the Unix kernel does, when you call chroot, is for that particular process, it changes how it evaluates /../ in this directory, so it basically changes /../ in /foo to point to itself, so it doesn't let you do this kind of escaping, and this change only applies to this process and not everyone else. Does that roughly make sense? So do you guys have any ideas about how you could escape a chroot environment because of the way it's implemented? Yeah.

AUDIENCE: So if you're [INAUDIBLE], you can make a directory and then bring that directory, and then go back to your directory and [INAUDIBLE].

PROFESSOR: Yeah. So, the interesting thing-- so the kernel only keeps track of one chroot directory. And I'll explain sort of the answer that I gave in a second. So what you could do is, maybe your chroot'd into /foo.

You're sort of stuck. You want to get at /etc/password, but how do you do it? Well, what you could do is you can actually open the root directory now. That will give you a file descriptor for effectively what is /foo. Then you could call chroot again. Maybe you can chroot into /bar. So now the kernel changes plan. Root is no longer /foo but it's /foo/bar and this /../ redirection only applies to /foo/bar/..

But know that you still have the file descriptor for /foo. So now what you could do is you could change directories into that file descriptor, fchdir(fd) from this open call, and now you chdir(..) And at this point, you were in /foo, you go to /../ from foo. It's no longer looped back to /foo itself, because you now have a different route and now you can escape, so this is perhaps a good illustration for why the exact mechanism matters a lot. It's not, sort of, the intuitive explanation that matters. And partly as a result, in Unix only the root user can invoke chroot, because otherwise chroot would be fairly pointless, in some ways.

So in Unix, you basically have to have uid 0 in order to chroot a process. It's a little bit of a disappointment in some ways, because if you wanted to build a really privileged separated system where everyone had just the minimum set of privileges necessary, you would probably need to use chroot, you would need to create new userids, et cetera, but in order to do that in Unix, you have to have a process running as root, which has lots of privileges. So it's a little bit of an unfortunate trade off, but it's probably one you could make some reasonable design decisions on. Question.

AUDIENCE: If in the [INAUDIBLE] directory, so [INAUDIBLE] to a file that's in [INAUDIBLE].

PROFESSOR: No, actually, unless you do this trick, the kernel evaluates symlinks in your root context, if you will. So if you have a symlink to /etc/password, it'll evaluate as if it is similar to /foo/etc/password.

AUDIENCE: [INAUDIBLE].

PROFESSOR: A hard link would not be protected. Yeah. So one way to set up a chroot environment without creating lots of copies of files is to, in fact, create a directory and hard link all these things back. That's fairly cheap and then use it.

AUDIENCE: If a program gradually generates inodes and, like, actually not to give you the file descriptor--

PROFESSOR: Right. So it's like a huge important detail here, is you can only access a file by path name, not by saying, I want to open inode number 23. This might be some weird file out there outside of my chroot. So in Unix you cannot open an inode by inode number unless you're root, of course. Other questions?

All right. So I think we have sort of enough machinery now to see what these OKWS guys do, and probably a useful, quick introduction is by contrast to what everyone else does. So what is it that everyone else is doing that the OKWS guys are afraid of? So the alternative design that pretty much every web server follows, is like the unprivileged separating picture above there, so you might have web browsers out there on the internet, these guys are going to connect to your server and, inside of your server, you're basically going to have, basically one process, httpd, well, Apache, let's say.

And this is one process running as a single userid called www in /etc/password. It takes all your connections, does everything with your process, including doing SSL processing maybe, including running application code and PHP, et cetera, all part of the same process. And if need be, this process will typically connect back to some database server, maybe MySQL could be running on the same machine, could be running elsewhere. And this MySQL process actually writes data to disk. But to connect to this MySQL, you probably have to provide a username and password but, typically, the way applications are written or, at least, not very security conscious applications are written, is that there's a single account on the MySQL server that the application knows the username and password for, so you just connect and you have access to all of your data.

So it's super convenient to write, because you just write whatever code you want. You can access whatever data in the database you want. There's no real isolation, but it has security problems that these guys worry about, namely if there's bugs in Apache, maybe in SSL, maybe in the application code or in the PHP interpreter, then inevitably the answer is, if there's a bug and you can exploit it, then, yep, you get the whole application data contents. Does that make sense? You had some questions before. No?

AUDIENCE: Oh. Uh, yeah, it's fine. Thank you.

PROFESSOR: All right. No worries. OK. So this is sort of state of art what these guys really wanted to protect against. And in their case, I guess they worried a lot because they were thinking, well, they're building basically a dating website, okcupid.com and they really wanted to make sure their, I guess reputation, wouldn't be damaged by these data disclosures.

So in fact, I guess-- from talking to the guy that wrote this paper, it seems like they actually haven't been compromised-- or, at least, not that they know of-- or their data wasn't leaked. And it seems to be partly as a result of running OKWS, partly as a result of maybe more proactive monitoring that they do, et cetera, but it seems to have worked out reasonably well for them, to some extent, I guess, because of this architecture that we have.

OK. So the reason that people, I guess, don't break up their applications into smaller components is because it actually takes quite a bit of effort to separate out all their pieces of code and define clean interfaces between them, decide which data every component should have access to, or if you decide to implement new feature, you're going to have to change the data that every component has access to give it new privileges or take some away, et cetera. So it's a bit of some overhead for separating application, but in their case I guess they decided it was worth the effort.

Let's try to understand how their web server design works, and perhaps one way to do it is to trace out roughly how http request gets processed by an OKWS server. So, similarly to that picture, there's probably a web browser out there somewhere that wants to go to okcupid.com, and in their design they sort of imagined they're going to have a bunch of machines, but we'll just look at probably just one front-end machine that is going to be running OKWS here and then another machine behind the scenes that's going to be storing the database somewhere. And I sort of imagine they're probably also using MySQL, because it's a nice piece of software in many ways. They don't want to re-implement this functionality, but they want to really protect this data, so that it's really hard to get to the raw disk or the raw database.

So how does a request work, or how does a request get handled by OKWS? Well, the request first comes in and gets handled by this process they call okd for the OKWS dispatcher. So those guys look at what the request is asking for and then actually does a couple things. So first it might need to log the request, so it forwards it to this component called oklogd, then it might need to generate some templates, maybe before the request came in even. And this is handled by another component called pubd.

And finally, there's a particular service that this request is being sent to, so okd has a table of a bunch of services it supports. This request is presumably going to one of them, and, as a result, okd will forward this request to a particular service process. And the job of the service is to actually do something with this request, like subscribe the guy to a newsletter or to match him to whoever else is using OkCupid, using the database.

And in order to do this, the service presumably might need to log some information about the request as well by talking to this oklogd component. And at the end of the day it's got to talk to this database. So the way these this guys actually implement talking to the database is that, unlike that Apache picture where you just talk to the database and issue arbitrary SQL queries, these guys come up with this notion of a database proxy that sits in front of the MySQL database and accepts requests from the service to do some queries, and I think that's most of the picture for OKWS.

There's another component in this whole picture that sparks this whole mess, so they have another component called okld, for the ok launcher demon, and this guy is responsible for starting all these processes on this front end web server machine. Hopefully some of these things actually look familiar, because this is exactly the architecture of [INAUDIBLE] for your lab assignment, so this is basically what our design is all based on. It seems like a nice design, actually. Well, we don't have pubd or logd, but we have these two guys and a service. No database proxy either. All right, so any questions about OKWS? Yes.

AUDIENCE: Dbproxy does not accept SQL queries, it accepts some sort

PROFESSOR: Yes. What does this interface look like? They don't really describe it in a lot of detail, but one thing I sort of imagine you could do in this database proxy is basically have this supply a bunch of arguments for SQL query templates. So it might be that this dbproxy, this one in particular, maybe is for finding, I don't know, your friends or something, so inside of the dbproxy maybe there's a template query like select ID from friends, where user-- I guess this is like the ID of the friend, and this is the ID of the person who is the friend of. The user equals, I don't know, person D here, or something, or person S here, right? And they sort of sanitize this string, and I imagine this RPC request here sort of looks like do query one, and the argument is, I don't know, "Alice." I sort of imagine this RPC interface looks like this, where the application knows ahead of time that this database proxy is willing to run three kinds of queries on its behalf, and now I want to run query number one and the argument is Alice. And that's sort of the way I get access to any data in the database. Does this make sense?

AUDIENCE: Could an external user at the web browser level send the request like that to the database or is that all internal?

PROFESSOR: Well, yeah. So, how does this work? It's actually kind of weird, that this is a separate machine, because now it seems like, why don't just connect to the database proxy yourself, or to the MySQL server, right? So what prevents this in their design?

AUDIENCE: Fire wall?

PROFESSOR: Yeah, probably at some level. They don't really describe this in too much detail, but probably this is some internal network, where there's like a switch here, and this machine is connected to the switch, this machine is connected to the switch, but the switch is not reachable from the outside world. It's like there is an internet connection here and those guys are some back-end network. Or maybe they're actually on the same network, but there's a firewall here that has rules that say, well, you can only connect to this front-end machine on port 80. You cannot talk to the back-end server.

So that's one plan. I guess the other plan they have in mind is that actually when you connect to this database proxy, you have to supply this 20 byte token thing, and unless you supply it, the dbproxy will reject your connection. So the rule is you open the TCP connection, you send your 20 bytes. If they're not the right 20 bytes, your connection gets closed, and hopefully this is something that's relatively easy for the database proxy to implement, so that there's probably low probability of a bug in that token checking logic that's right up front. And unless you have the token, you will not be able to do anything else of interest to the database server. That's, I think, their sort of design goal here. Make sense?

All right. So, let's try to figure out, I guess, how these guys isolate these different processes. So how do they make sure that all these components don't trample on each other? What's the plan?

AUDIENCE: Different roots and different userids?

PROFESSOR: Yeah, so pretty much every one of these components runs as a difference uid, so they have this whole table in the paper that describes, for every component, where is it running and what's the uid. So we can write this out, so okd has its own uid, pubd has its own uid, the logger has its own uid. okld runs as root, which is kind of unfortunate, but may be all right. Then, there's a whole bunch of dynamically assigned userids for every service. I sort of imagine he has ID 51001.

So this makes sure that every service cannot poke at the processes of other services, and they also use chroot pretty extensively, so every one of these guys is chroot-ed into some directory. They sort of initially say, well, you should really chroot everyone into a separate directory. As it turns out, in that table, it turns out that okd and all the services basically share a chroot directory. It's kind of weird. Why do you guys think they put okd and the services into a single chroot and not give them their own chroots? Weird. Yeah.

AUDIENCE: okd is not root.

PROFESSOR: Well, yeah, but like why don't they put pubd and oklogd and everyone else in the same chroot as well?

AUDIENCE: Okld [INAUDIBLE].

PROFESSOR: okld is actually sitting out here in a separate chroot. This guy, this guy. Actually okld is not chroot. I'm sorry about that. These guys are chroot. Does it matter?

AUDIENCE: I was thinking, if the services have to share a lot of data, where's the [INAUDIBLE] isolate them?

PROFESSOR: Maybe. Actually, I think what's going on is they have to share some data, but none of this data actually lives in the file, so they pass a lot of data through sockets from okd to the services. But in fact, none of these guys store anything of interest at all in the file system.

So as a result, there's pretty much nothing interesting in the chroot directory, so I imagine the OKWS guys just decided, well, there's probably some overhead to creating a chroot, like you have to create a copy of the directory, there is maybe some management overhead for every chroot. Whereas, for this, there's no real files here, so maybe that's all right. I mean, there's not a clear, sort of, cut trade off here, or not a clear-cut argument which way you should go, but certainly prevents from setuid binaries. The reason that these guys are probably in different chroots is because there is actually some interesting stuff there. There's maybe the templates here, maybe there's a log file here, so you don't want these guys accidentally reading the log file for some reason, but there's no real mutable state inside of the chroot shared by all okd and the services.

AUDIENCE: Don't the services have, like, two files or, I don't know, aspx files.

PROFESSOR: Well, at least the way they describe it in the paper, the service is a single C++ compiled binary, so there's actually no extra files, and there are templates, but those actually get passed in through this weird mechanism, where pubd has the templates in its directory, it renders them-- or sort of pre-computes them-- sends them to okd and okd gives the templates to all the services through RPC calls. So they sit in memory, but they're actually not directly accessible through the file system. It's a somewhat paranoid design here. Can't even read the templates.

So what's the point of having all these components split up? So I guess let's talk about maybe oklogd, so why do you have a separate oklogd? Yeah.

AUDIENCE: Because if you had [INAUDIBLE], you could overwrite logs, or truncate log.

PROFESSOR: Yeah, so we really want to make sure that if something goes wrong, the log, at least, is intact. So there's a separate log file that is only writable by this uid, and all the log messages are sent as RPCs to this log service, and even if everything else gets compromised-- well, except for okld-- then the log is still intact, because they talk about the process of appending noise to this log. So what is this about? Does this matter? Should we worry about this noise? Yeah.

AUDIENCE: If somehow you accidentally find a way to read the log and you can't see what everyone else has been doing?

PROFESSOR: No, I think this noise thing they are actually worried about is that, suppose you compromise a service or you compromise pubd or something, you all of a sudden might be able to write log messages to the log, and you can actually write whatever you want to the log at that point. So the only guarantee they claim to provide is that, I guess, before the point of the compromise, all the log entries are intact, and afterwards there is sort of legitimate log entries interspersed with whatever else the attacker wants to log.

One actually cool thing about having oklogd be a separate process instead of it just being an append-only file, is that oklogd cannot add some extra information to each log entry, because you could imagine maybe the operating system supports an append-only file, but then you don't actually know who wrote anything to a file, when that was, whereas oklogd, for every message, it can actually maybe time stamp it and say, actually, I know this came from this service, or this came from okd, so you actually get extra information in that log file because it's a separate service here. Make sense?

So what's the point of this okld guy? Why do we need this guy to be running as root? I guess a couple reasons.

AUDIENCE: If you want no one else to run as root, then you need okld to delegate who it is.

PROFESSOR: Yes. So someone needs to set up this whole uid chroot thing, and you need root for this in Unix, so okld is it. That's one reason. Anything else?

AUDIENCE: To define 80?

PROFESSOR: Oh, yeah, yeah. This like whole listening on a port business, you have to bind on port 80, so okld does that as well for us. Anything else?

AUDIENCE: Complete to open the log file oklogd. You don't want to open oklogd to have access to open the file. You want to open it for [INAUDIBLE].

PROFESSOR: Maybe. Actually, yeah. I don't know. I forget this from the source code whether they actually do this or not. You could imagine absolutley--

AUDIENCE: I think they write it in the paper as well.

PROFESSOR: I see. So okld opens the log file and passes it in? Could be.

AUDIENCE: Because otherwise an attacker that compromised oklogd would be able to erase the entire log.

PROFESSOR: That's right, yeah. So maybe you want to open in append-only mode and then pass it to oklogd and then you have more security guarantees for the log. Yeah, that's actually pretty cool. I missed that on the paper, but makes a lot of sense, yeah. Any other things okld is doing for us? I think that's basically it. These are the main things that you can't do unless you're root, and okld is sort of the component that ends up having to do all these operations.

So I guess we had this homework question about what happens if you leak this 20 byte database token thing. So what do you guys think? What's the damage? Should we leak these guys? Should we worry about it? Anything else?

AUDIENCE: The attacker can pretend to be that specific service [INAUDIBLE].

PROFESSOR: That's right, yeah. So, you might be able to now connect and issue, of course, all these template queries. That actually seems fairly straightforward, I guess, from this picture. You probably need to compromise one of these components to be able to connect to the database server in the first place. So I guess if you have this token and you manage to compromise one of these pieces in the picture, then you could run all these queries as well. Make sense? Fairly straightforward stuff.

OK, I guess let's look at, could you do better? Could you do better than this OKWS design? Except for make this whole argument about, well, we might be able to do even better, like allocating a separate unit uid per user in this design instead of per service. But here, every service, like newsletters or friend matching or account sign up is a separate userid, but every OKWS user isn't really represented by a Unix uid. There's not really userids, they're service IDs. So, would it make sense to have different uids for every OKWS customer? Is there a reason for that? Yeah.

AUDIENCE: So at the moment, if one user compromises the service, then they can get access to all the other user's data for that same server.

PROFESSOR: That's right, yeah.

AUDIENCE: Whereas, if you had a separate-- essentially a separate service and a separate dbproxy for every user, there's no way you could access anyone else's data either.

PROFESSOR: Right, but could it be actually a stronger model? So especially for-- well, I guess there's really two reasons why I think the OKWS guys don't go to that extreme model. One of them is-- they make a big deal in this paper-- is performance, right? So if you have, I don't know, a couple million users on OkCupid, then all of a sudden you have a couple of million processes running here, or maybe a couple million dbproxies, or maybe you can optimize something on the dbproxy side, but here, yeah, you have a couple million userids and either you have a lot of processes running all the time or you're starting these processes on demand. And starting a process involves some nontrivial amount of overhead, so you probably wouldn't be able to get as good of performance numbers as these guys are able to show with OKWS. There's a performance argument. Question.

AUDIENCE: Yeah, I was just reading in the paper that said the performance of the system was better than others?

PROFESSOR: Yeah.

AUDIENCE: How come?

PROFESSOR: Well, I think it's partly because they fine tuned their design to their particular workload and it's also they write their whole thing in C++. The alternative is you're writing some stuff in PHP, then you're probably going to win on that front. It's also the case that they don't have nearly as many features as, let's say, Apache has.

Apache has a very general purpose design, so it has lots of processes running, it restarts them every once in awhile. It actually has every ttp connection tying up a process for the duration of that connection. They do keep-alives. That also increases the number of processes you have to run for their design. So all those things just add up in terms of overhead for Apache, because it wants to handle anything possibly you could do with a web server. Whereas these guys, I think, are very specific. We're just going to run these services, very quick requests, and no even static file serving, if they can help it.

But I think there's actually other web servers out there these days that probably can match the performance of OKWS if you really wanted to. So, for example, Nginx is a very optimized web server you can run these days. If you want fast application performance on the server side, you probably want to keep a long running process very much like the OKWS service thing, and the fast cgi is a common mechanism, or sort of protocol, that you could use on the server side to implement this even in Apache or Nginx as well.

So I think many of these performance ideas aren't exclusive to OKWS. You couldn't perform the same performance tricks in other services as well. They just show that better security doesn't preclude these tricks. You could still get the performance.

I guess for them, they were just initially starting out with an Apache-like design, where they were willing to pay the price if it was easy to program and secure, but it just wasn't secure, so they said, OK, well we'll do this, and I don't think the performance was as necessarily a big of a goal for them. I guess they wanted to-- well, at the time they had some problems in terms of performance as well, so I think they really wanted good performance. Any other questions about this stuff?

OK. So I guess it was saying, one reason why these guys don't want to run a separate service per user is the fact that there's performance overheads in doing that. The other reason is that their full application model sort of hinges around a service having to get access to every user's data, like finding your friends on

OkCupid or someone to go out on a date with.

And as a result, this per user isolation model might not make a lot of sense, because ultimately there has to be a service that you're going to send a request to, and it's going to look at everyone else's data to find your match. So that's probably, as a result, not really amenable, like, even if you had user IDs or some sort of per user isolation mechanism, you would have to give that service access to every userid anyway.

So for other services like maybe Gmail or Dropbox, where it's much more user focused and no sharing, then a per user isolation might make more sense in terms of the benefits you could get out of it, because if there's a userid on the Dropbox server for every Dropbox customer, then, well, there's a process running for you and there's a process running for someone else, and if you exploit a bug, then you can't touch other people's data.

But could be cool, I don't know. Dropbox probably doesn't do this for performance reasons, but you could get some security benefit. Whereas, for OKWS guys, and even functionality-wise, they wouldn't be able to take advantage as much of this model. So maybe for your profile editing service, maybe that could be run per user, but the matching thing would still be shared. Make sense?

All right. So let's look at whether OKWS actually manages to improve security here. So when we do think of whether a system is secure or not is to look at all the components and see, well, first of all, what's the attack surface? Meaning, how would you try to compromise that component, or how hard is it, and second what's the damage? So let's go through this list.

Let's start with okd, so what's the attack surface? What kinds of things could you use to attack it? [INAUDIBLE], like all these requests coming from the browser. That seems pretty good. You can control it probably in lots of ways, you can send lots of strange input.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah, so maybe there is-- this thing is written in C++, so probably these guys were sloppy. I think this guy's a good programmer, but if he was not very careful somewhere, could be easily exploitable. So what's the damage? Suppose you find a buffer overflow or some other bug in okd? How bad is this? Yeah.

AUDIENCE: You can call basically any service on that machine?

PROFESSOR: Yeah, OK, so you could call any service. Is that bad? How should we think of this?

AUDIENCE: You can call it whatever input you want.

PROFESSOR: That's true, yeah. But you could probably have done that even without compromising, because you can send any http request you want, which is basically what these service requests end up being. Maybe that's actually not so bad. Yeah.

AUDIENCE: Could it route all the traffic for maybe OK website?

PROFESSOR: Yes. That actually seems a little more damaging, right? You can all of a sudden take over the whole website and serve your pages instead of sending a request to the services. You could redirect all the people to match.com or whoever else you want to, I don't know. I guess now they bought out OkCupid but, before, who knows. OK, anything else? Could you leak data in any way? Yeah.

AUDIENCE: Well, it depends on, like if you use any authentication in okd. Instead of any SVC, you could potentially just do unauthorized requests, like in the database?

PROFESSOR: Right. In their case, this guy just parses and forwards the request on.

AUDIENCE: Could it [INAUDIBLE]?

PROFESSOR: Yeah. Not only can you redirect it first, you could actually look for all the subsequent requests, which probably includes passwords of other users connecting to the site, and you could save their passwords or modify their requests or see what they're doing or fetch things on their behalf. That seems damaging potentially. That's probably the biggest leak. If you compromise okd, you can probably watch other requests and steal people's credential passwords, steal their data as it flows by.

Make sense?

AUDIENCE: Could you do some sort of denial of service with just with sending a whole--

PROFESSOR: Yeah, so you could probably chew up the CPU or send lots of requests to this, fill up database with lots of data, but that you could probably do even by just sending lots of requests in the first place. So now [INAUDIBLE] are somewhat complicated-- well, different. Yeah.

AUDIENCE: So, the goal here is not to leak data--

PROFESSOR: Yeah.

AUDIENCE: --from different services. If you have access to okd, presumably you could read the responses that are being sent--

PROFESSOR: Yeah, exactly. So in fact, okd is has to be pretty trustworthy, because the responses don't go directly back through okd in the normal operation, because you just pass off the fd, and the service directly writes to the fd. But you could totally fake it and create your own fd here. So if you compromise this, you could basically watch all the traffic and steal people's passwords there.

AUDIENCE: But, the other way--

PROFESSOR: And the response is--

AUDIENCE: --to the output, which means that essentially you get access--

PROFESSOR: That's right. Yeah, so you could-- If okd was compromised and I happened to log into that site, you could probably look at my responses and you could probably even take my password and send other requests with my credentials and get data from there as well. Yeah.

AUDIENCE: And then essentially reconstruct the entire database.

PROFESSOR: Exactly. Yeah, or at least for users who were logged in at the time and things you could reconstruct. This is pretty damaging, right? So this component is actually a bit troublesome here. What happens if we compromise, let's say, oklogd? How bad is that? Yeah.

AUDIENCE: [INAUDIBLE] pretty bad.

PROFESSOR: Yeah, so there's like all this confidential data in the log entries, then that's probably not good, but otherwise you can't probably access the database directly, right? Pubd, I guess you might corrupt templates, or something like this, send out different requests and responses, I mean. Yeah.

AUDIENCE: So this is about the logd. Presumably okdlogd doesn't have access to read the log. It just needs--

PROFESSOR: Ah, yeah, yeah. Good point. Well, it depends on how they do it, right? If they really have this append-only file, then it might be that you can't even read the log, so--

AUDIENCE: All you could do is append garbage.

PROFESSOR: That's right, yeah. So you could write a lot of garbage, but if they were using the OS to enforce the append-only log, and no reading, then you might be actually in good shape for the log contents. Yeah.

AUDIENCE: Well, I guess also, you can not append. So when a valid log comes in--

PROFESSOR: Right, so you could probably block real entries, fill it up with garbage. You could also watch new entries and at least compromise them.

AUDIENCE: Or if you're relying on a rate limit for number of logins.

PROFESSOR: That's right, yeah. You could probably do that. OK. So what about the services? That's, I think, their main attack factor, because-- actually, in most of these systems, what you're really worried about is the one-off components, because even in

Apache, the Apache code is probably pretty good.

Like millions people are running it, everyone's looking for bugs in it. Probably [INAUDIBLE] not that many bugs in Apache itself or, well, even in SSL, for all the hoopla we've heard recently about [INAUDIBLE] and still bugs probably not as bad as the application code that you write for a particular site, because no one else has reviewed that code. You just wrote it, you haven't really tested it very thoroughly. That's probably where most of the bugs in a complex system actually lie. So the service code is probably the equivalent for OKWS.

These components are written by Max Krohn. He was careful to make sure there's no buffer overflows. This component is written by some web developer who wants to deploy the next feature as fast as possible, so this is the part where I think they really worry about bugs being sort of exploitable and potentially damaging. But hopefully the damage here is not too big, in the sense that you can only issue whatever queries you are allowed to do by the database proxy. Make sense?

So what about okld? This is a bit of a sore thumb here. It's, like, running as root. How much should we worry about it? Of course, the damage is pretty big if we compromise it. You get access to everything on the machine and all the database proxy tokens. How hard is it to compromise okld? What signals-- what could you poke it with? Does it take input?

AUDIENCE: [INAUDIBLE] in a very specific pattern.

PROFESSOR: Yeah, so pretty much the only input it takes is when a child exits and it gets a notification that a child process exited and then maybe it responds it, or not, if it is rate limited. So if there's some sort of erase condition or a bug with handling exactly-- it's interesting-- lots of exits at the same time, then maybe you could trigger something bad, but even then it seems hard to imagine injecting some sort of shell code through the exit pattern. So it's probably a reasonable thing to have to run as root, because it doesn't take a whole lot of input. Make sense? Other questions?

AUDIENCE: So presumably a big concern would be if you managed to somehow exploit the dbproxy.

PROFESSOR: Uh-huh.

AUDIENCE: If it turns out that it doesn't, like, it provides an RPC that's limited in scope, but if there is some input you can give that, that turns out to run in different query than it was expecting to run, presumably that could be a big [INAUDIBLE].

PROFESSOR: That could be a bit of a problem. So what's the attack vector though on this database proxy? I think you have to have access to one of these other components in the first place. So at least you have to now compromise both-- you have to find a bug both in the dbproxy and somewhere else, so--

AUDIENCE: So not necessarily, because the SVC is already forwarding--

PROFESSOR: Right. So as the SVC passes through great query is largely unchecked--

AUDIENCE: Well, so, I mean, let's say you're trying to log in, right? It will pass.

PROFESSOR: Yeah, like your Alice name goes in the template, and yeah.

AUDIENCE: Alice is straight to the dbproxy, in theory.

PROFESSOR: Absolutely. So, there might be some dbproxy bugs here that are also exploitable. Anyway, this hopefully gives you guys some sense of how do you think of privileged separating application. And as we see, it's not perfect. There are still many things that could go wrong. But it seems much better than the nonprivileged separated design that we started out with.