Revised Transcript - Lecture 14: Security and Protection

All right.

Welcome everybody. Low attendance today, I don't know what is going on. Anyway, last day we wrapped up. We talked about the Multics model. The main thing I want you to see in the Multics model is this idea of escalating our permissions from one set of permissions up to a more enabling set of permissions, which we do by going through this idea of a gate. Where a gate is a specific function entry point where we can take care to make sure that whoever is escalating the permissions gets suitable attention to make sure that they should be escalating their permission. And the other thing in the Multics world is this idea that if we want to go down to a domain that has fewer permissions, we do that, but we copy our data so they've given us this sort of basic set of tools so that we can think about what should happen as we escalate and deescalate our permissions.

OK so, as we mentioned last day, Multics never got completely built. Instead, we have things like Unix, which is an outgrowth of the Multics project, and we now have lots of things that are outgrowth of the Unix projects, including Windows. So everything in the Microsoft world that comes after Windows NT, the so-called new technology originally called Cairo, is a derivative of Unix. Tiny bit of history here. It's technically a derivative of a Unix version 6. It is technically a derivative of an operating system project called Mach, which was a distributed version of essentially a Unix, so you'll occasionally see the word Mach pop up. If you're on a Mac machine and you look at an executable type, you'll see it reports all of your executables as type Mach.O, and if you read any of the stuff about the original design of what is now the main development pattern followed by Microsoft, it's all based on Mach, but these are all based in turn on Unix. So when we talk about Unix domain strategies, we're really talking about the thing that is providing the foundation for pretty much all of the other operating systems that are likely to be used.

OK, so I think last day we wrapped up. I learned that you did not actually have a discussion prior to this course about user level permissions on files. So we talked about user, group and other bit flag permissions. These are sometimes called the mode. If you want to change them, there is a program called Change Mode which is spelled CHMOD, so a lot of people will call it chmod. This allows you to set the permissions on a given file so you can set something to be non-writable, also known as read-only. You can change that permission so that you could increase or decrease the permission for other users, etcetera. So this is an example of user-based restricted domains.

So in the modern operating system world, we have this idea that you have multiple different domains that can be associated with a given entity, such as a program or a file, and file system permissions are one of these types of domain. So that is a set of restrictions that we can control at the file system level on any entity in the file system, which can then have an influence, for instance, on whether we're allowed to run a given program if it is or is not executable.

In the Unix world, we have one account that is allowed to do anything. So that account we call root. The root account has global permission to set any permissions, including for itself, and if you can set any permissions, including your own permissions, then you have all permissions, because if you didn't have them originally, you could just set them and award them to yourself. So root exists so that it can solve problems. The idea of the root account is that the root account always has permission to set or revoke anyone else's permissions for any other task, and can therefore, by extension, run any programs, do any work required.

This is different than the supervisor bits that we set for the kernel, because the kernel isn't really running as a user. The kernel's running before we have users. So when we boot the machine, the first thing we do is we load the kernel into memory and set the kernel running. Once the kernel has done all of the setup work that it needs to do, such as probing for devices, configuring any disks, checking file systems, all this sort of thing. Once the kernel is satisfied that the computer is running properly, it will, in the parlance they use in the Unix world, "go multi-user." And at that point, what it does is it starts running user processes.

So one of the first user processes it runs is a process called INIT and the init process then runs all of the other processes including things like login so that you as a user can then log into the system. So init will be running as root. It's the first process that gets run. Right. So it is created by the kernel, it's really the only process that doesn't get created by calling fork. Because it runs everything else, so init runs all of the other processes. Init starts as root, but the kernel you can think of as sort of this almost like primordial prehistoric thing before the time begins, when you can start running user programs, the kernel has to be fully up and running. Its last step of the boot process is to start the user space programs running by running init. Init then runs as root. Init and its children and grandchildren and great grandchildren are all the things that you think of as the running programs on the system. They can run as different user accounts, but there are no user accounts before you have something that can actually read the user account database. So therefore the kernel is really separate from the entire user space.

OK, it is running as the process that got booted and it's set up user accounts later, so you can think of it as if you needed to associate a user account with it. You could think of it as essentially having all permissions and therefore kind of is like root, but root is really a user account for user space programs that get run later. Root account because it has all permissions, can do anything. It can delete any file. It can create any file, it can change any file, it can run or destroy any program. But I again want to draw the distinction between that and the supervisor bit kernel which not only can juggle all the running programs, but it can reconfigure their memory space. The kernel can do things like talk to the device subsystem. So no user program, regardless whether it's root or not, can talk to the device driver code without first invoking the kernel through a trap.

OK, so these are different than at the hardware level. You have supervisor bit kernel, non-supervisor bit everything else. In the permission domain world, we have root which is just a special account that has all

permissions, but on the outside world of user space programs. The inside world of the operating system is the kernel, where it can do things like change who's got what memory mapped, reconfigure people's page tables, decide what IO device is going to get called next, and it does things at a hardware level, none of which are available to the user program.

OK, so the root account is to provide a user program level facility to manage the system. If you need the kernel to be doing something different, let's say you need to change the kernel configuration so that it is going to have a fundamentally different way of operating. How do you do that? Let's say you wanted it to have a new device. You need to install a new device driver. How are you going to get the kernel to understand the new device driver? Fundamentally, you need the kernel to have different things in it. How do you get that to happen? Have you ever installed a device driver? There's an important last step in installing any device driver or similar things. You reboot the system, you get rid of the old kernel. The old kernel's no longer running because you rebooted. Now you're booting a new kernel that is going to wake up and be able to talk to the new device driver code, take advantage of the new configuration. Now you can talk to that snazzy new device you just bought, a scanner or I don't know, printer or whatever it is that you got. So the reboot happens so that the kernel can be different. Root is in charge of everything once the kernel's going, so root can do things like change users on the system. You can add a user, you don't need to reboot to do that. You've changed some configuration files. You have permission to do that as root. But you didn't need to change the whole kernel because you're changing things at that user level, not at the hardware level. So as soon as you're at hardware level, we're going to need to reboot to get a new kernel if we need to change any of the configuration.

OK, so this is a super user, root. It can do anything any other user can do and more, including create and destroy users and their files and anything else. There's an additional bit, the set user ID bit. So if you change the set user ID bit of an executable file, which you can only do as the user owning the file or as root, then that means that any user who runs that program, their identity upon running the program gets switched to the user who owns the file. This is pretty slick, so this allows us to do permission escalation. So you can, if you were brave enough and trusted the program and us, you could write a program that did some cool thing, and if it was owned by root, you could then as root set the user ID bit, and that means that any user who runs that program effectively becomes root during the running of the program. So that program could then do things that no other program could do because it's now running as root, it has the ability to read or write anyone's directory. It can reconfigure anything on the system. That's a lot of power.

Typically we use this in a more restricted way. So you would have a set user ID, but you wouldn't set it to root. You'd set it to some other user that has a different set of permissions. So this is how we get things like the web server running as user www. The web server is set up with set user ID so that when you run it, it turns into user www. It can do all the www stuff, and more importantly, it can't do the non-www stuff like, I don't know, delete your disk or steal user accounts or anything else that you don't want it to do. So we can use this set user ID to change our identity. And by getting the identity changed, we change our

permission domain. And the Unix plan is that we do this managed by the file system interface. So the two things that we have in the file system interface to control that is the file owner, so that's the user ID who is listed as the owner of the file, and then we can have this extra bit set to say change the user identity when the program's run, which means the thing you're setting this on has to be a program or it's pretty useless. If you just have like a text file readme.txt and you change the set user ID bit on it, well, you've set a bit, but it's not going to have any effect because when you edit the file, it's still just a text file, not going to turn you into somebody else.

So the set UID protocol is based on this idea that when you run the program and the set UID bit is set on the program, your identity changes when you run the program. And so because that is very dangerous, you don't want to be able to write a program and randomly turn any user into anyone else. You're only allowed to set the user ID bit on a program that you own, or if you are root. So you could write a program so that your friend could do something as you. Presumably you trust the code that you wrote, which is why you're willing to set the user ID bit on this, because it's a program, right? You should know what it does. You can read the source code, you can figure out what that program is actually going to do. If the program has a well understood restricted set of operations that it does, and you trust anyone to run that program under your user ID domain, you might set user ID bit on them. As root, you're allowed to set not only the user ID bit but the ownership. So you could say I want to make this program owned by so and so and anyone who runs it is just going to run it under their user account. User root is pretty powerful in that way. Does that make sense to everyone? You see what I'm talking about?

I guess here I can give you a quick example. Don't know if I have a terminal going here. Let's get a terminal up. So if I go to... Go into my A4 solution because I know... Nope, that won't work. Most of the program here. Yeah, well, no, that's not going to work either. Where do I have a nice program? Let's just make one. Everybody's favorite example program.

OK, so I have two files, both owned by me. So now I have set the user ID bit, which, because they didn't want to make the display any wider, they've just overridden on top of some of the other bits. So a lowercase s in the third position means that you can have a set user ID bit set. If I was to do the same thing to hello.c, you can see it shows you with a capital S to say that the set user ID bit is kind of foolishly set on the source code file to really very little effect, but it is set. So if anyone was on this computer other than me and wanted to run hello, they would turn into me as they were running. That's pretty safe because I know that hello.c didn't do anything particularly dangerous, so I'm not giving away any great permissions by letting someone become me by running that. Cool.

OK. So a daemon process which we mentioned last time is simply a process whose parent exited. So it is no longer attached to the command line. Daemon processes provide a lot of services to temporarily do work as a different user ID by providing a network interface. So essentially what this allows you to do is pass a message through a thing we call a socket. So that's very much like a typical byte stream that you

would have if you opened a file, only it will pass over the Internet. You're using a socket every time you use any kind of networking code where you're attaching to a process on another server.

So you're probably aware that every time you look at a web page, you're supplying a machine name. This is an identity on the Internet. You're further supplying what we call a port number, which is an identifier. You can think of it as like a channel on the far machine. Web servers run on port 80, so you've essentially established a byte stream as though you opened a connection to the process listening on port 80 on the distant machine, and you send them some bytes and they can send it back to you, and you read and write exactly like you would with any other file interface. This allows a generalized way of talking to a program which has registered itself with a machine name and a port number to get some kind of work done, and you're effectively asking the other program to do work for you. Which means the other program doesn't have to be running as your user account. You can therefore essentially say, hey, would you do this work for me? Like look up some web pages or print a file or whatever the other task might be and it doesn't need to be across the Internet. You can ask for someone on your own machine to do some work for you in the same way. So this idea of having a daemon process running under some user account and the ability through the network to send it a message to say please do this task for me means you can get that task done under that user account.

OK. The next one here is the funnily named chroot. This stands for change root. So this is a tool that has been around in Unix for a while. Quite a while really. And what happens is when you say I want to change the root of the file system to some specific location in the command passed to chroot as an argument, it looks like the file system of the computer only exists from that point down. OK, so if you chrooted a particular command, let's say a shell, even, to have as its new root directory your home directory, then it wouldn't be able to see that there was anything else on the system. It wouldn't know about the temp directory, wouldn't know about the bin directory, wouldn't know about any of the other user directories. So this is a way of saying this command is allowed to run, but I'm going to run it in this restricted environment. It can only see the files from that part of the file system and the children of that part of the file system. But that's the only restriction. Otherwise, it's running exactly the same as the other programs on the system, it just has this kind of mirage of what the file system looks like.

There's an extended version of this that comes from the BSD world where they said, well, that sounds like a good idea, but we need it to be more restrictive so that people can't, you know, try to attack other things on the system by, let's say, talking to daemons. So a jail is simply an extension of this chroot idea where we say not only am I going to restrict where you can see in the file system, but I'm also going to restrict what network connections you can have. I'm going to restrict what programs you're allowed to run, and I'm going to restrict what users you're allowed to run things as, so it's essentially a more restrictive version of what's happening with the chroot setup.

So you can see I've got a note here, BSD family only for about 20 years. Linux has been promising they'll add the jail. They haven't really gotten around to adding the jail yet. They have now started referring to chroot as a chroot jail, which is very confusing to the rest of the community because they used to be very distinct things. But now they're muddying the waters with the terminology. The chroot is literally just the file system. The jail has these additional tools. There's starting to be less interest, however, in the Linux community in completing the work of the jail, and this is because now we have full virtualization. So instead of having a chroot jail, what they'll do is they'll start up something heavy, like a container such as Docker, where you essentially have a whole fake computer running as a program inside your computer. We don't have time in this course to talk about virtualization, but if you take 3050, we'll talk about it there at length. The only downside of the Docker style plan is that it's really heavyweight, takes a lot of resources to run a Docker style container, whereas these two, chroot and jail, are incredibly lightweight. They require almost no overhead, so you can run literally thousands of jails on a very low end kind of server and not notice it, whereas you're going to be able to run a very low number of Docker containers before you start noticing your machine would slow down.

OK. Unix gives only a few permission restrictions within the running process. So this is where they said the Multics ring plan is great, but too complicated for us. Programmers are not generally that interested in having different permission domains inside of the same process, if you have access to the ability to have other processes. So the Unix model is essentially, if you want two different parts of your code running as two different users, just have two different processes. They can communicate with shared memory and semaphores. You can have one running as one process, one running as another process. You can move data back and forth easily. You don't need to say, "Oh, once I enter this function I would like my whole permission domain to change." So the only thing that we have inside of a running process in the Unix model are these things: system calls give you a fundamentally different permission to make you become the kernel. Right, you trap into the kernel. The kernel can do anything. Not only can it rewrite file systems, it can rip apart whole running programs. There's no restriction. It literally has raw access to all of memory and all the devices. There's no way you can restrict anything the kernel can do. So system calls to go into and return from kernel access is one of the main permission restrictions we have so that user programs do not have the access that would enable them to accidentally destroy the system. Right. You cannot destroy someone else's virtual memory context because you literally can't see it. You're in your own private virtual memory. Only the kernel has access to everyone else's virtual memory.

Historically, in the page table we had a read-only bit that we could set on a page. So this allowed us to say this page should not get written. So therefore, if someone tries to write to this page that is an invalid operation, you're going to get a bus error or segmentation violation or something of that nature, and that user process is going to be removed. We've now added an execute bit. Why would we add an execute bit on each page? This basically says are you allowed to run instructions from this page. If you try to load something from this page into the instruction register and execute bit's not set, you will again have a segfault.

Your stack is not part of the text segment. There's no reason instructions should be on the stack, so this was one of the first strategies to try to prevent worms. Got instructions loaded on the stack and instruction register set to a location on the stack? Boom, you were yanked out of the run queue. You were no longer running. Much better than having a worm on your system. Now, as I mentioned, we also have the random offset for the segments because we have so much memory space available, so they also make it difficult to even find the stack, but the execute bit really helps.

OK, so on Windows, similar. So the FAT or NTFS permissions are a subset of the Unix permissions. We don't have full group, we don't have full world, but under NTFS we do have read, write, execute permissions for the individual files. The root, which is an account, you can actually log in as root. You can't do anything without assuming the root account identity, so this is what su or sudo does. Instead of this we have administrator permissions and properties. So you can set administrator property on any user account. There also by default is an account whose name literally is administrator that has the administrator property on it, and if you want to be really confused, remove the administrator property from the administrator account and you'll have lots of fun.

Yeah, any questions regarding the differences between chroot and jail?

Just which person? You can see directory set as the new root. There is no way to go back up. Same way as if you're on the root of your current file system and you try to go up. There's just no further up to go to, but a chroot is just literally that. It's just a file system restriction. Jail adds the ability to restrict other things such as access to the network. There's additional tools where you can talk about what kind of which users are allowed to run things, etcetera. So it's just additional permission restrictions that you can set in a jail.

OK, so Unix root is an actual account. Windows Administrator is a set of properties you can set on an account. There is no real distinction made between the system and an administrator accounts. And it's been very messy for many years and is only slowly getting better. The assumption in the Windows world has been that anyone who's doing important work probably is doing it as administrator. Many of the developer tools don't even work properly if you're not running as an account that has administrator privileges, which is a terrible plan. Because if you're running as administrator at any time, you can accidentally destroy the entire system. The whole point of the separation is so that you're not running as the super user when you're not doing things that need to be done as the super user.

OK, so if you're on a Unix machine and you need to configure the system, you consciously say I need to change to be the super user so that I can change the system configuration. On a Windows machine, all too often people will say, "Oh I can't run my compiler so I have to set administrator privileges on my account," and then next day you're running Word as root. If you have a macro virus, it's running as root. If you want to surf a web page, you're running as root. You have permission to destroy everything all the

time. This is one of the reasons Windows is perennially attacked by various Trojans and viruses, and so on, 'cause they built this nice separation system and then took the separation down by sloppily saying, "Oh, we'll just let anyone who's doing any kind of serious work." Where serious work involves running a compiler. I mean, really, what does the compiler do? It reads some files, it writes some files. There's absolutely no reason that a compiler needs to be run as administrator, except somebody decided they were going to set it up so it didn't work properly if it wasn't run as administrator. We'll come back to that in one second to see how they do that. So this is a very unfortunate history that until they finally get past, we're just going to see Windows machines perennially hacked by viruses and Trojans and everything else until they clean up their act. Sad but true.

OK, so in the file system we always have, as we saw in the FAT assignment, there's a hidden bit, so that just means the file browser doesn't show it to you by default. There's a somewhat equivalent of the hidden property under the Windows or under the Unix world. You know what it is? You say LS, it does not list all of the entries in the current directory. You know which ones it omits? Anyone? OK. And what is the first character of the previous directory? It is a dot. So every directory has an entry whose name is dot which is myself. It has one whose entry is dot dot which is my parent. The root's parent is just the root, loops around, but by default LS omits printing anything whose name begins with a dot.

So if in your home directory on Linux or any other Unix machine, and you say LS minus A for all, you will see a whole bunch of things that you never saw before. There's a bunch of directories there and files whose name begins with dot. You might know some of them. You've got a dot bash RC and a dot bash profile, and there's a dot SSH directory, and so on. And because they start with dot, LS doesn't show them to you. They're perfectly fine entries. LS just skips them because it skips things that begin with dot. The Windows plan is that you literally set a bit on that entry to say I don't want to see it. So it could have any name. You can just turn on or off the hidden bit. There's also a read-only or non-writable bit, and there's a bit where you can flag it whether it is a system file which has a lot to do with whether the file browser and other tools let you play with it or not.

So I mentioned a couple of times the so-called new technology or Cairo project resulted eventually in Windows NT that is the basis of all of the really serious versions of Windows that have come along. Windows actually started, if you went and got the original Windows, it was a DOS program. It was about faking out multiple DOS programs into thinking they could run at the same computer at the same time. There was no memory protection. There was very little other than they set a different base address in memory to load the program to, so they were forever stepping on each other's toes. But it did mean that you could with care load and print from a word processor while, say, using Excel, but anything more complicated than that, it kind of fell over.

Many editions later we had for a while the Windows Home editions. You'll still hear these talked about occasionally. Those were still based on the no-memory-protection model. The main issue was that if you

had certain types of memory problem. So if you had a program where it was writing where it shouldn't be writing. If you had no other program in memory, it would run fine under DOS or the Windows home models because there was no one to hit. It could write to the wrong location and as long as the wrong location didn't hit something within your own memory space, you could put it there. It was illegal, but you could put it there. If you move that to the NT system, it said segment fault. You're writing to an invalid memory location.

This is why they staggered along for so long with the two versions of Windows. You can still put a Windows 11 executable into DOS mode, which basically means it allocates a full memory space in the virtual world, to make it look like an empty DOS machine, so you can still supposedly run some of these old, badly written programs and they'll stumble along. We don't like that in general. We like programs that are well written, so we have some process page permissions in the NT specification, but although they had lots of these permissions including execute bit, they only were honoring read-only until we got as far as Windows XP. So that's about 10 years after they put them in. They didn't even use the execute bit stuff.

OK, returning to why you have to be administrator to run a lot of the programming tools. They have put the configuration information in Windows in a thing called a hive. You're allowed to have multiple different types of hives on your Windows machine, but you always have one global hive that they call the registry. So the registry is a fancy key lookup tool.

So if you think about it, let's say a hash table. You're used to a hash table where you've got keys and values. You store something under a given key. You probably use these in languages like Python, where they're called things like dictionaries. OK, also frequently called an associative map, so the general process is that you have a key and you have a value and you store the value under the key and later you come back with the key and you can find the value. The hive is that plus the tree structure. So they've got this idea that your keys are in a hierarchical organization, so essentially there's a prefix to each key that matches other keys of some related type. And they've got this idea that you can access something at a prefix level and get a list of all the keys that have the same prefix.

So there's a couple of top level prefixes. One of them is HKEY_CURRENT_USER, which is where all of the user profile stuff for the currently logged in user is stored. So you can find out all the stuff about your information in the HKEY_CURRENT_USER part of the registry. If you log in as a different user, you'll still have an HKEY_CURRENT_USER part of the registry, but now it's been updated for the new user that you're logged in as, and so you can find out things like what is my shell, where is my home directory, all the stuff which is associated with a given user. The user is allowed to read and write all these things, so this is where you keep your path, all of your environment variables. They're all in this part of the registry.

There is also one called HKEY_LOCAL_MACHINE. So this is the same type of information, but for configuration on this physical computer. Because they've got it set up, but you can have this as a

distributed property so you can have the same user identity on multiple different Windows machines that are under the same administration envelope, but they might have different local machine properties because they may physically be different local machines, right? They may even have different versions of Windows on them. And then there's a bunch of others. Everything except the current user is protected from a non-administrator user. It may be so protected you can't even read the values.

This is where they got sloppy with the compiler because in the compiler you need to know important things like what is my instruction set architecture, which is stored in HKEY_LOCAL_MACHINE. And rather than make sure that things that were important to the compiler were at least readable by all users, they went the lazy route and said, well, we'll just make you be administrator as a compiler user so that you can read the things to find out what machine you're on. Still hoping they're going to recover from that.

OK. Going back a little bit to the more generic stuff. One of the ways these are stored, in both Windows and in Linux, is using what we call an access matrix. So all we do here is we say if I've got different permission domains, I can, for a given permission domain, so let's say a particular user, I can talk about all of the different resources available on the system, and I can have the set of permissions relative to that resource. So I can look up in my matrix what's going on. Sure you can. You can store things in a matrix. I think everybody understands how that works.

The thing that is interesting about access matrices is that this also allows us to map who's allowed to switch from one permission domain to another. So a lot of them will say, so domain D2 is allowed to switch to D1. It's not allowed to switch back to itself. But it is allowed to switch to D3. By contrast, D1 isn't allowed to switch to anyone but D3, so this says I can essentially change my permissions. I'm not going to say escalate because we can't really tell whether we're escalating or deescalating, but we can set up a list of legal changes so then D2 can become D3, D3 can become back to D2, D2 can become D1. D1 can only become D3.

We generally specify whether we can have, when doing data transfers, whether we're doing a copy, whether we're transferring ownership when we do one of these switches or whether we're doing a limited copy, which basically means we're allowing the thing we're switching to to have access to the data, but it's not transitive, they're not allowed to pass the data further along. The other way you'll see these implemented is rather than having a matrix, which you'll notice has lots of gaps in it - gaps are bad for memory management because it's an empty space that you have to store somehow. So the other way you'll see this done is instead of a matrix, just as a list, which is just a list of values associated with each domain. You've got a different list for each domain and that saves potentially a lot of space if you have lots of different permission sets. You need to store, but it wastes a lot of time because now you can't do direct mapping. You have to do linear search because it's a list.

OK, so in summary. In theory, we can assign permissions to users. Permissions can protect users against each other. Permissions can protect the system against the users. But in practice, implementation lags.

We have some tools to do this, but they're not as flexible as the theory might imply. So one of the things we have under Linux, in order to allow flexible and finely-tuned permission strategies associated with particular needs, we have the older model, the so-called pluggable authentication modules or PAM. Anyone who's played around a lot with SSH may see these, so you can add PAM modules to your SSH configuration to do various things such as support different key types, allow different types of access once you've logged in, so on and so forth.

They then added loadable security modules, so these are actually kernel modules that you can load at boot time to change what different permission interfaces your kernel is going to provide to your user land utilities. So loadable modules, again a thing we'll talk about in 3050 if people want to take that. But loadable modules are essentially additional pieces of the program text that we load after we've got the main part of the program loaded. Right. So additional pages of instruction that we can add in to a program once it is running. So one of the implementations of these that you're probably familiar with are dynamically loaded libraries or shared objects, and the loadable security modules in Linux are another type of implementation of that. They also now do loadable modules for things like device drivers, where they can load some of the device drivers after they've got the kernel running.

OK, in Windows we have a security ID to check domain with resources. And these have a discretionary security control by default, which means that it isn't necessarily enforced by the hardware. It is essentially a suggestion that various tools are supposed to abide by. So a lot of the security implementation is based on requesting "Am I allowed to do operation X?" You get an answer back and then you decide to do X based on the answer, but it may not be the case that you're actually physically prevented from doing X, even if the system gives you the answer no.

So it's essentially based on the program behaving properly according to what the permissions are telling it it can or cannot do. That's what we mean by discretionary security here, so much weaker than what we might think of as real, heavily enforced security, where you're simply prevented from doing the illegal action. OK, so this solves our problem of restricting resource access, but doesn't necessarily ensure that we have full control over what trusted entities are actually doing.

So remaining problems. One of them is what we call confinement. So confinement is about data escape. So if you've got data in a particular domain and you have an entity that can enter that domain and run code within that domain, there's no way you can prevent that entity from giving the data away. This is how essentially every data breach works is that you've got your data carefully protected, but somehow someone got a program running in there and the program gave away all the goods. Once the goods are given away, they're gone, right? You can't get a copy of the data back. You can't be sure it's destroyed. There's nothing we can do the way we have designed modern computers to prevent data from being able to be copied, it's just bytes. So at the end of the day, if someone can get a copy of the pattern of bytes that

you have that are your secret bytes and let them move out of your restriction domain, you've lost confinement.

OK. All of our strategies, they suffer from the super user issue. Sometimes people call this the God Mode problem. One user has all control. So that user is the point of attack. Single point of failure. Anyone who can cause a process to become super user or can cause a super user process to run something on their behalf? Well, now they've bypassed all your security. Doesn't matter how many gates and locks you have. They have all control. They're allowed to do anything they want. So having that super user account or permission is by design a single point of failure for security on your system.

We also have a problem with messaging. It's very hard to get programs to do anything important without IO. And however you're doing the IO becomes itself a place where an eavesdropper may be able to get a hold of your data. So there is no defined, guaranteed secure way to communicate without the potential of eavesdropping. At some level, something can always interfere with user-user communication. If you think about it, the whole design of the kernel means the kernel is a single point of failure here. Right. If a user process puts some secret bytes in a buffer and wants to, I don't know, use a pipe or shared memory or anything else to communicate it to another entity, well, the kernel had to set the buffer up, so you have to be trusting the kernel. There's no private communication without some underlying potential security leak.

So in general, we would like to be able to say I want to access data in only a trustworthy environment. And I would like to be able to totally prevent any untrustworthy code or users from having access. But the problem is that the definition of trust isn't defined by the entity who has to make the trust agreement. If you are running code on the computer, you are effectively believing that the system administrator's definition of trust is good enough for your purposes. The only way you can get around that is if you only run code on your computer where you are also the System Administrator, but it's pretty hard to do work that way. Right. We're in the age of the cloud. The cloud is always in fact, famously has been described as "someone else's computer." So you're essentially saying whatever their definition is of trustworthy, I'm willing to go by their definition because they're in control in that environment. And so all I can do is I can deploy programs there, but I can't change the trust environment associated with that setup. Really, we want trust to be able to be managed by our programs that we're running in that environment. No one offers that at the current time.

OK, so some ways around some of these problems. One of them is cryptography. So we have a whole course on this. I will talk about it very briefly here. Cryptography literally comes from the Greek "writing in code." We actually use the words encode and decode to talk about how to get your data in to or out of the cryptographic format. So when we encode the data, or encrypt they sometimes say, we're using what we call a cipher. The cipher is an algorithm to take some data and rewrite it in a hidden form.

Modern cryptography uses a key. So that you have some key that is used in conjunction with your cipher. Remember, the cipher is an algorithm, so the cipher will take your data and the key and combine them.

Your data is in this case referred to as the plain text. This is the readable version of the data, maps it into a cipher text. So if you're talking about, let's say, secure shell, it's taking all the data that you type at the terminal and mapping it into this cipher text, sending it across the Internet, and then at the other end decrypting it to turn it back into the plain text so that the remote machine can find out what you're trying to type. We also do this with, let's say, messages, so you can do cryptography on e-mail messages, you can do various types of things.

The plan is you've always got point A where you have a trust environment. You take your plain text and you make it secret. You send it out through the dangerous wide world over to point B so that somebody in the middle can't look at the ciphertext and guess what you really typed. They usually talk about Alice at point A and Bob at point B and then Eve the eavesdropper hanging out in the middle trying to guess what your secret message was, stealing all your goods. That's called cryptanalysis, where without the key and maybe even without knowledge of what the cipher is, they're trying to guess what the message was as it went by.

Simple ciphers can be decoded with frequency analysis, so one of the earliest ciphers was what they called the Caesar cipher, where you just take all the letters in the alphabet and you rotate them. So let's say you add five. Right. So A becomes E, B becomes F, etcetera. That's easy to guess because if you have, let's say, English language messages, some letters occur a lot more than other letters. Does anyone know what the letter is in English that occurs the most often? Anyone? A vowel? Yeah. Letter E. So if someone is just taking the alphabet and rotated it up, if you have enough text, you can just count and say symbol 57 is happening an awful lot. That must be E, and if everything else just shifted then once you know E you know everything else. So the Caesar cipher is not particularly secure. Even more or somewhat more complex ciphers still frequency analysis can uncover them. If you read any wartime literature they're always talking about breaking the various ciphers during various past wars. And this is essentially what espionage is all about. So spies are interested in capturing ciphertext as it goes by and figuring out what was actually in it.

OK, so there's two main families. We have symmetric, where you want to recover the plain text afterward. So you have your secret key, you encode it. Your friend better have the same key. They're going to use that key to decode. The key is a secret, so you have a problem of how to get the key to your friend without sending it past the eavesdropper. Somehow you need to get the key securely to the other end. Then you're in good shape. If your friend has the key, they can decode, they can get to the original.

The other use that we have is what we call hashing and/or one-way encryption. So the secret key is used to hash things, but we're not interested in unhashing. A common use of this is passwords. So we store passwords on the system after they've been one-way encrypted. So you cannot decrypt the password file. What you can do is start guessing a password. So what you do is you take a password. Someone gives you their password to put in the file. You add a couple characters of salt. So the system just produces some salt, it picks 2 characters. And then it puts the password into, if you're on a Unix machine, DES encryption

algorithm, which is an algorithm family that was developed by the US government in the 60s. And you store the encrypted results and the salt.

So then if someone else comes and supplies a password and you want to figure out whether they should be allowed in the system, you repeat, you take the salt, you take the thing they gave you, you encrypt it using the same algorithm, and you check whether the encrypted result is the same encrypted result as the one you put in the password file. This allows us to have a nice compact storage. It's actually a many-to-one mapping. This is one of the reasons we can't go backward is in theory, there are some other passwords that could possibly be typed that mapped to the same password you have, but the space is so big, the likelihood of someone actually typing that by accident, especially using only the keys that are available on the keyboard, 'cause we're talking about byte values now, so we're including byte values like null and 253 and delete and other things that are not necessarily valid parts of the password. They could in theory map to your password. In reality, we don't worry about this because out of the set of passwords you can actually generate as characters sent in to the terminal, the likelihood of multiple passwords mapping to the same hash code is pretty close to 0.

So we store this and then even if someone gets the password file, they can't figure out your password. They do then have the salt. So the one thing they could do is start taking a dictionary of all possible words that you might have used as your password and start hammering them into the password file with the salt to see if they get the same encrypted results. This is called dictionary attack, and is a type of password cracking.

I will say that I think I told you before I did my undergrad here and around that time was when they first started providing central login e-mail to students. And someone actually stole the password file, so then they wanted to know how good the student passwords were, and they found that a remarkable number of students had as their password the word "secret". And they discovered that this was because they gave each of the students a piece of paper that said your password must be "secret". These are people who are not particularly savvy computer users, so they were just following the instructions they got from computing services. They were then all asked to change their passwords pretty quickly. I don't know why they put the quotes on the piece of paper. That was just silly, but they never figured out why they put the quotes there either.

So then, as I said, each time you apply the password you re-encrypt it using the salt that you put in the file and then it's just a simple string compare. So an intruder has to guess all the passwords in turn. If they don't have the password file, they also have to guess the salts, so they're essentially just submitting through login. And this is now made even more secure than it used to be because now we layer file system permissions on this, so the password file which obviously needs to be accessible to the login program so that it knows, or sorry, the password file which needs to be accessible to anyone who wants to be able to see what users are on the system. So if you run LS minus L and you need to know what your

user ID is, you need access to the password file for logins but not for password purposes. Password data itself only needs to be accessible to the login program which is running as root.

So what we now do is we have two password files, one of which has the passwords as well as all of the other login information like your user identity login string, your user ID number, your group number, your home directory, all that stuff. The other one has all that plus the password. So as a regular user, you can look at on Linux machine you can look at, etc/passwd and you'll see all of the password file information, but the password field is empty. You have to be root in order to be able to get to the actual shadow password file that contains the data.

OK. So then ramping up security? Different environments obviously give us different security. Java was actually written with the intention of making a highly secure environment. This was one of the very early goals of the Java virtual machine. Just having a virtual machine gives you the ability to completely change the rules of what's happening inside of the runtime environment for anything in that machine. You essentially get to say I don't need to only rely on the trust environment of the operating system. I'm going to build my own trust environment in the virtual machine. It's more restricted than the one outside.

Every routine in a JVM is isolated from each other. You can actually set up for every method and every function entrance and exit data inspection on the entry and exit from a particular block of code. All the data gets allocated from a private heap that no one has access to. And every class can have its own security domain. So you're going to have a Java program with multiple classes in it, and define which classes are allowed to do what things. This class is allowed to write a file. That class isn't allowed any file access whatsoever. So if you're inheriting or if you're supporting modules you can have security domains associated with the classes coming from those and restrict what's going on.

The big one, though, was the applet sandbox, which Java has pretty much outgrown. The applet world, the vision was that this is how we would write active or dynamic web pages. For whatever reason, JavaScript has taken that over, which by the way is a huge promotional win, I guess. JavaScript has almost nothing to do with Java other than it was also written by Sun and around the same time, and Java got popular and the JavaScript people wanted to cash in on Java's popularity. So they put the word Java in the name of their programming language. JavaScript has very little security at the beginning, but they got some good press because they were also called Java.

So if you run a Java applet, this is essentially a program in an even more restrictive environment, so it's a program that you found because you encountered a web page. So it's fundamentally from someone else's machine. You've gone to a web page, you access the web page, on the web page is source code which you're going to run inside of a JVM locally. And you're going to be able to trust that because it's running as an applet, which means it has its own private sandbox created by the JVM on your machine. You're running random Internet code locally, but in this box that doesn't allow any access to any files on your machine. And further does not allow any network communication to any computer other than the

original server that you got the code from, so it can't stage an attack on the Internet. It can't stage an attack on your machine. It can't destroy any of your files. The worst it can do is have an infinite loop and use up some of your CPU time because the IO is so constrained.

Java also has a lot of type safety. I'm sure you've all - have you all taken the Java course? By this point you've done some Java. It's very annoying doing all the literal casting you have to do, so they do that on purpose because they want to make sure you know what you're converting into what else. There's also stack inspection, so you can if you want to increase your privileges, you can block a call to a method or a function to say I want to do this in a privileged way. I'll show you that in one second. And you can also have a whole security policy that you as the user can enforce on a JVM that you're running. So before you run some Java code, you can actually have a security policy file that says I'm willing to run this program, but only under these restrictive conditions. And it could only do this because it's an interpreted language within a virtual machine where we're trusting the virtual machine to actually enforce the security. So it is much more secure than, let's say, a C program, which is just running natively in whatever security environment the OS provides.

OK. So then to do this kind of do privilege check permission? So if we have an untrusted applet and it wants to call a URL loader, so it wants to, let's say connect to something in the uoguelph.ca domain. Then it needs to do that using a socket. So we have a socket permission as an applet. It has no permission whatsoever to do the socket access so to increase our permission, we're going to have to say... So at the GUI level in the applet, it's going to call get. Get with a URL is going to then enter the URL loader permission domain associated with the URL loader module. The URL loader is going to say I would like to do in a privileged fashion an open of the address uoguelph.ca. If that succeeds, then we do the URL access work. If that fails, we're going to get an exception and we're going to teleport out of get back to wherever the exception handler here is. So I haven't put a try-catch around this, so this would just keep going up the stack until we get to main, if it is not caught somewhere else.

So open with an address is then going to say check the permission on the address based on whether it is allowed to have connect permission. URL loader has connect permission so therefore this call to open because it is done inside of a doPrivileged block from a module that has that permission will be allowed to connect. Notice that I do the URL work and then I come back and I call open again. What's going to happen here? Are we going to be allowed to continue? So the reason this one was allowed is because it was in the doPrivileged block, so it was. This code is declaring "I'm aware I'm asking for a special privilege." And it has the permission class connect so that when we check the permission for connect based on the call to open it succeeds. This one is going to fail because it's just directly calling open. So we're going to come here into the networking core. Right. The call to this function succeeds. We come into this function, but when we ask has a permission been requested based on the permission connect from this call to open, the answer is no, because it's coming from a permission domain with no permissions. And it's not doing it in a doPrivileged block. So this is simply going to say get out of here and

it's going to raise an exception and boom, we're going to see this fail and it's not going to give us access to open that particular connection here, even though by going through the get interface to call open we were allowed to.

So the reason we were allowed to here is because in the JVM the URL loader module has been configured to have the connect permission. And because it is signaling that it knows it's asking for extra permissions. OK, so by having the doPrivileged block from a module that has the right permission, when checkPermission says does this object have the permission connect, the answer is yes. If we were in the URL loader and we didn't do the doPrivileged block, it would still fail because when we said did someone ask for permission for connect, the answer would be no. This was not done within a doPrivileged block. So the Java security hinges on this - does the module have, has it been authorized to ask for this permission? And is it aware that it is asking? The assumption being that if you have code that was naively written and does not understand that it is doing something dangerous, the answer to "may I do the dangerous thing" should probably be no. So you can declare "I know I'm about to do something I wouldn't normally be allowed to do. And I have been stamped, I've been certified for connect permission" therefore checkPermission for connect permission will succeed. Anyone else asking when we in the networking core code where it's doing the dangerous thing and we don't want anyone at all just calling it, we see whether the caller in fact has the permission. Makes sense?

OK, so that is very timely. 2 minutes left. We just finished the security module. And with that, the content for the course. So I hope you enjoyed the content of the course and learned a lot of great things. On Tuesday, we'll do review. What have I asked you to do to prepare for the review? Come up with questions, yes, so I will, I meant to do this last day, I will add a forum where people can put up some questions and we can therefore harvest some questions from online. But bring your questions on Tuesday as well and we'll see what we have to say, yeah. So 4.