55

Access Control

Chapter by **Peter Reiher (UCLA)**

55.1 Introduction

So we know what our security goals are, we have at least a general sense of the security policies we'd like to enforce, and we have some evidence about who is requesting various system services that might (or might not) violate our policies. Now we need to take that information and turn it into something actionable, something that a piece of software can perform for us.

There are two important steps here:

1. Figure out if the request fits within our security policy.
2. If it does, perform the operation. If not, make sure it isn't done.

The first step is generally referred to as **access control**. We will determine which system resources or services can be accessed by which parties in which ways under which circumstances. Basically, it boils down to another of those binary decisions that fit so well into our computing paradigms: yes or no. But how to make that decision? To make the problem more concrete, consider this case. User X wishes to read and write file /var/foo. Under the covers, this case probably implies that a process being run under the identity of User X issued a system call such as:

open("/var/foo", O_RDWR)

Note here that we're not talking about the Linux `open()` call, which is a specific implementation that handles access control a specific way. We're talking about the general idea of how you might be able to control access to a file open system call. Hence the different font, to remind you.

How should the system handle this request from the process, making sure that the file is not opened if the security policy to be enforced forbids it, but equally making sure that the file is opened if the policy allows it? We know that the system call will trap to the operating system, giving it the opportunity to do something to make this decision. Mechanically speaking, what should that "something" be?

THE CRUX OF THE PROBLEM:
HOW TO DETERMINE IF AN ACCESS REQUEST SHOULD BE GRANTED?

How can the operating system decide if a particular request made by a particular process belonging to a particular user at some given moment should or should not be granted? What information will be used to make this decision? How can we set this information to encode the security policies we want to enforce for our system?

55.2 Important Aspects Of The Access Control Problem

As usual, the system will run some kind of algorithm to make this decision. It will take certain inputs and produce a binary output, a yes-or-no decision on granting access. At the high level, access control is usually spoken of in terms of **subjects**, **objects**, and **access**. A subject is the entity that wants to perform the access, perhaps a user or a process. An object is the thing the subject wants to access, perhaps a file or a device. Access is some particular mode of dealing with the object, such as reading it or writing it. So an access control decision is about whether a particular subject is allowed to perform a particular mode of access on a particular object. We sometimes refer to the process of determining if a particular subject is allowed to perform a particular form of access on a particular[1] object as **authorization**.

One relevant issue is when will access control decisions be made? The system must run whatever algorithm it uses every time it makes such a decision. The code that implements this algorithm is called a **reference monitor**, and there is an obvious incentive to make sure it is implemented both correctly and efficiently. If it's not correct, you make the wrong access decisions – obviously bad. Its efficiency is important because it will inject some overhead whenever it is used. Perhaps we wish to minimize these overheads by not checking access control on every possible opportunity. On the other hand, remember that principle of complete mediation we introduced a couple of chapters back? That principle said we should check security conditions every time someone asked for something.

Clearly, we'll need to balance costs against security benefits. But if we can find some beneficial special cases where we can achieve low cost without compromising security, we can possibly manage to avoid trading off one for the other, at least in those cases.

One way to do so is to give subjects objects that belong only to them. If the object is inherently theirs, by its very nature and unchangeably so, the system can let the subject (a process, in the operating system case) ac-

[1] Wow. You know how hard it is to get so many instances of the word "particular" to line up like this? It's a column of particulars! But, perhaps, not particularly interesting.

cess it freely. Virtualization allows us to create virtual objects of this kind. Virtual memory is an excellent example. A process is allowed to access its virtual memory freely[2], with no special operating system access control check at the moment the process tries to use it. A good thing, too, since otherwise we would need to run our access control algorithm on every process memory reference, which would lead to a ridiculously slow system. We can play similar virtualization tricks with peripheral devices. If a process is given access to some virtual device, which is actually backed up by a real physical device controlled by the OS, and if no other process is allowed to use that device, the operating system need not check for access control every time the process wants to use it. For example, a process might be granted control of a GPU based on an initial access control decision, after which the process can write to the GPU's memory or issue instructions directly to it without further intervention by the OS.

Of course, as discussed earlier, virtualization is mostly an operating-system provided illusion. Processes share memory, devices, and other computing resources. What appears to be theirs alone is actually shared, with the operating system running around behind the scenes to keep the illusion going, sometimes assisted by special hardware. That means the operating system, without the direct knowledge and participation of the applications using the virtualized resource, still has to make sure that only proper forms of access to it are allowed. So merely relying on virtualization to ensure proper access just pushes the problem down to protecting the virtualization functionality of the OS. Even if we leave that issue aside, sooner or later we have to move past cheap special cases and deal with the general problem. Subject X wants to read and write object `/tmp/foo`. Maybe it's allowable, maybe it isn't. Now what?

Computer scientists have come up with two basic approaches to solving this question, relying on different data structures and different methods of making the decision. One is called **access control lists** and the other is called **capabilities**. It's actually a little inaccurate to claim that computer scientists came up with these approaches, since they've been in use in non-computer contexts for millennia. Let's look at them in a more general perspective before we consider operating system implementations.

Let's say we want to start an exclusive nightclub (called, perhaps, Chez Andrea[3]) restricted to only the best operating system researchers and developers. We don't want to let any of those database or programming language people slip in, so we'll need to make sure only our approved customers get through the door. How might we do that? One

[2] Almost. Remember the bits in the page table that determine whether a particular page can be read, written, or executed? But it's not the operating system doing the runtime check here, it's the virtual memory hardware.

[3] The authors Arpaci-Dusseau would like to note that author Reiher is in charge of these name choices for the security chapters, and did not strong-arm him into using their names throughout this and other examples. We now return you to your regular reading...

way would be to hire a massive intimidating bouncer who has a list of all the approved members. When someone wants to enter the club, they would prove their identity to the bouncer, and the bouncer would see if they were on the list. If it was Linus Torvalds or Barbara Liskov, the bouncer would let them in, but would keep out the hoi polloi networking folks who had failed to distinguish themselves in operating systems.

Another approach would be to put a really great lock on the door of the club and hand out keys to that lock to all of our OS buddies. If Jerome Saltzer wanted to get in to Chez Andrea, he'd merely pull out his key and unlock the door. If some computer architects with no OS chops wanted to get in, they wouldn't have a key and thus would be stuck outside. Compared to the other approach, we'd save on the salary of the bouncer, though we would have to pay for the locks and keys[4]. As new luminaries in the OS field emerge who we want to admit, we'll need new keys for them, and once in a while we may make a mistake and hand out a key to someone who doesn't deserve it, or a member might lose a key, in which case we need to make sure that key no longer opens the club door.

The same ideas can be used in computer systems. Early computer scientists decided to call the approach that's kind of like locks and keys a **capability-based system**, while the approach based on the bouncer and the list of those to admit was called an **access control list system**. Capabilities are thus like keys, or tickets to a movie, or tokens that let you ride a subway. Access control lists are thus like, well, lists. How does this work in an operating system? If you're using capabilities, when a process belonging to user X wants to read and write file `/tmp/foo`, it hands a capability specific to that file to the system. (And precisely what, you may ask, is a capability in this context? Good question! We'll get to that.) If you're using access control lists (**ACLs**, for short), the system looks up user X on an ACL associated with `/tmp/foo`, only allowing the access if the user is on the list. In either case, the check can be made at the moment the access (an `open()` call, in our example) is requested. The check is made after trapping to the operating system, but before the access is actually permitted, with an early exit and error code returned if the access control check fails.

At a high level, these two options may not sound very different, but when you start thinking about the algorithm you'll need to run and the data structures required to support that algorithm, you'll quickly see that there are major differences. Let's walk through each in turn.

[4]Note that for both access control lists and capabilities, we are assuming we've already authenticated the person trying to enter the club. If some nobody wearing a Linus Torvalds or Barbara Liskov mask gets past our bouncer, or if we aren't careful to determine that it really is Jerome Saltzer before handing a random person the key, we're not going to keep the riffraff out. Abandoning the cute analogy, absolutely the same issue applies in real computer systems, which is why the previous chapter discussed authentication in detail.

55.3 Using ACLs For Access Control

What if, in the tradition of old British clubs, Chez Andrea gives each member his own private room, in addition to access to the library, the dining room, the billiard parlor, and other shared spaces? In this case, we need to ensure not just that only members get into the club at all, but that Ken Thompson (known to be a bit of a scamp [T84]) can't slip into Whitfield Diffie's room and short-sheet his bed. We could have one big access control list that specifies allowable access to every room, but that would get unmanageable. Instead, why not have one ACL for each room in the club?

We do the same thing with files in a typical OS that relies on ACLs for access control. Each file has its own access control list, resulting in simpler, shorter lists and quicker access control checks. So our `open()` call in an ACL system will examine a list for `/tmp/foo`, not an ACL encoding all accesses for every file in the system.

When this `open()` call traps to the operating system, the OS consults the running process's PCB to determine who owns the process. That data structure indicates that user X owns the process. The system then must get hold of the access control list for `/tmp/foo`. This ACL is more file metadata, akin to the things we discussed in the chapter titled "Files and Directories." So it's likely to be stored with or near the rest of the metadata for this file. Somehow, we obtain that list from persistent storage. We now look up X on the list. Either X is there or isn't. If not, no access for X. If yes, we'll typically go a step further to determine if the ACL entry for X allows the type of access being requested. In our example, X wanted to open `/tmp/foo` for read and write. Perhaps the ACL allows X to open that file for read, but not for write. In that case, the system will deny the access and return an error to the process.

In principle, this isn't too complicated, but remember the devil being in the details? He's still there. Consider some of those details. For example, where exactly is the ACL persistently stored? It really does need to be persistent for most resources, since the ACLs effectively encode our chosen security policy, which is probably not changing very often. So it's somewhere on the flash drive or disk. Unless it's cached, we'll need to read it off that device every time someone tries to open the file. In most file systems, as was discussed in the sections on persistence, you already need to perform several device reads to actually obtain any information from a file. Are we going to require another read to also get the ACL for the file? If so, where on the device do we put the ACL to ensure that it's quick to access? It would be best if it was close to, or even part of, something we're already reading, which suggests a few possible locations: the file's directory entry, the file's inode, or perhaps the first data block of the file. At the minimum, we want to have the ACL close to one of those locations, and it might be better if it was actually in one of them, such as the inode.

That leads to another vexing detail: how big is this list? If we do the

obvious thing and create a list of actual user IDs and access modes, in principle the list could be of arbitrary size, up to the number of users known to the system. For some systems, that could be thousands of entries. But typically files belong to one user and are often available only to that user and perhaps a couple friends. So we wouldn't want to reserve enough space in every ACL for every possible user to be listed, since most users wouldn't appear in most ACLs. With some exceptions, of course: a lot of files should be available in some mode (perhaps read or execute) to all users. After all, commonly used executables (like `ls` and `mv`) are stored in files, and we'll be applying access control to them, just like any other file. Our users will share the same font files, configuration files for networking, and so forth. We have to allow all users to access these files or they won't be able to do much of anything on the system.

So the obvious implementation would reserve a big per-file list that would be totally filled for some files and nearly empty for others. That's clearly wasteful. For the totally filled lists, there's another worrying detail: every time we want to check access in the list, we'll need to search it. Modern computers can search a list of a thousand entries rather quickly, but if we need to perform such searches all the time, we'll add a lot of undesirable overhead to our system. We could solve the problem with variable-sized access control lists, only allocating the space required for each list. Spend a few moments thinking about how you would fit that kind of metadata into the types of file systems we've studied, and the implications for performance.

Fortunately, in most circumstances we can benefit from a bit of legacy handed down to us from the original Bell Labs Unix system. Back in those primeval days when computer science giants roamed the Earth (or at least certain parts of New Jersey), persistent storage was in short supply and pretty expensive. There was simply no way they could afford to store large ACLs for each file. In fact, when they worked it out, they figured they could afford about nine bits for each file's ACL. Nine bits don't go far, but fortunately those early Unix designers had plenty of cleverness to make up for their lack of hardware. They thought about their problem and figured out that there were effectively three modes of access they cared about (read, write, and execute, for most files), and they could handle most security policies with only three entries on each access control list. Of course, if they were going to use one bit per access mode per entry, they would have already used up their nine bits, leaving no bits to specify who the entry pertained to. So they cleverly partitioned the entries on their access control list into three groups. One is the owner of the file, whose identity they had already stored in the inode. One is the members of a particular group or users; this group ID was also stored in the inode. The final one is everybody else, i.e., everybody who wasn't the owner or a member of his group. No need to use any bits to store that, since it was just the complement of the user and group.

This solution not only solved the problem of the amount of storage eaten up by ACLs, but also solved the problem of the cost of accessing

and checking them. You already needed to access a file's inode to do almost anything with it, so if the ACL was embedded in the inode, there would be no extra seeks and reads to obtain it. And instead of a search of an arbitrary sized list, a little simple logic on a few bits would provide the answer to the access control question. And that logic is still providing the answer in most systems that use Posix-compliant file systems to this very day. Of course, the approach has limitations, since it cannot express complex access modes and sharing relationships. For that reason, some modern systems (such as Windows) allow extensions that permit the use of more general ACLs, but many rely on the tried-and-true Unix-style nine-bit ACLs[5].

There are some good features of ACLs and some limiting features. Good points first. First, what if you want to figure out who is allowed to access a resource? If you're using ACLs, that's an easy question to answer, since you can simply look at the ACL itself. Second, if you want to change the set of subjects who can access an object, you merely need to change the ACL, since nothing else can give the user access. Third, since the ACL is typically kept either with or near the file itself, if you can get to the file, you can get to all relevant access control information. This is particularly important in distributed systems, but it also has good performance implications for all systems, as long as your design keeps the ACL near the file or its inode.

Now for the less desirable features. First, ACLs require you to solve problems we mentioned earlier: having to store the access control information somewhere near the file and dealing with potentially expensive searches of long lists. We described some practical solutions that work pretty well in most systems, but these solutions limit what ACLs can do. Second, what if you want to figure out the entire set of resources some principal (a process or a user) is permitted to access? You'll need to check every single ACL in the system, since that principal might be on any of them. Third, in a distributed environment, you need to have a common view of identity across all the machines for ACLs to be effective. If a user on `cs.ucla.edu` wants to access a file stored on `cs.wisconsin.edu`, the Wisconsin machine is going to check some identity provided by UCLA against an access control list stored at Wisconsin. Does user `remzi` at UCLA actually refer to the same principal as user `remzi` at Wisconsin? If not, you may allow a remote user to access something he shouldn't. But trying to maintain a consistent name space of users across multiple different computing domains is challenging.

[5] The history is a bit more complicated than this. The CTSS system offered a more limited form of condensed ACL than Unix did [C+63], and the Multics system included the concept of groups in a more general access control list consisting of character string names of users and groups [S74]. Thus, the Unix approach was a cross-breeding of these even earlier systems.

ASIDE: NAME SPACES

We just encountered one of the interesting and difficult problems in distributed systems: what do names mean on different machines? This **name space** problem is relatively easy on a single computer. If the name chosen for a new thing is already in use, don't allow it to be assigned. So when a particular name is issued on that system by any user or process, it means the same thing. `/etc/password` is the same file for you and for all the other users on your computer.

But what about distributed systems composed of multiple computers? If you want the same guarantee about unique names understood by all, you need to make sure someone on a machine at UCLA does not create a name already being used at the University of Wisconsin. How to do that?

Different answers have different pluses and minuses. One approach is not to bother and to understand that the namespaces are different – that's what we do with process IDs, for example. Another approach is to require an authority to approve name selection – that's more or less how AFS handles file name creation. Another approach is to hand out portions of the name space to each participant and allow them to assign any name from that portion, but not any other name – that's how the World Wide Web and the IPv4 address space handle the issue. None of these answers are universally right or wrong. Design your name space for your needs, but understand the implications.

55.4 Using Capabilities For Access Control

Access control lists are not your only option for controlling access in computer systems. Almost, but not quite. You can also use capabilities, the option that's more like keys or tickets. Chez Andrea could give keys to its members to allow admission. Different rooms could have different keys, preventing the more mischievous members from leaving little surprises in other members' rooms. Each member would carry around a set of keys that would admit him or her to the particular areas of the club she should have access to. Like ACLs, capabilities have a long history of use in computer systems, with Dennis and van Horn [DV64] being perhaps the earliest example. Wulf et al. [W+74] describe the Hydra Operating System, which used capabilities as a fundamental control mechanism. Levy [L84] gives a book-length summary of the use of capabilities in early hardware and software systems. In capability systems, a running process has some set of capabilities that specify its access permissions. If you're using a pure capability system, there is no ACL anywhere, and this set is the entire encoding of the access permissions for this process. That's not how Linux or Windows work, but other operating systems, such as Hydra, examined this approach to handling access control.

How would we perform that `open()` call in this kind of pure capabil-

ity system? When the call is made, either your application would provide a capability permitting your process to open the file in question as a parameter, or the operating system would find the capability for you. In either case, the operating system would check that the capability does or does not allow you to perform a read/write open on file `/tmp/foo`. If it does, the OS opens it for you. If not, back comes an error to your process, chiding it for trying to open a file it does not have a capability for. (Remember, we're not talking about Linux here. Linux uses ACLs, not capabilities, to determine if an `open()` call should be allowed.)

There are some obvious questions here. What, precisely, is a capability? Clearly we're not talking about metal keys or paper tickets. Also, how does the OS check the validity of capability? And where do capabilities come from, in the first place? Just like all other information in a computer, capabilities are bunches of bits. They are data. Given that there are probably lots of resources to protect, and capabilities must be specific to a resource, capabilities are likely to be fairly long, and perhaps fairly complex. But, ultimately, they're just bits. Anything composed of a bunch of bits has certain properties we must bear in mind. For example, anyone can create any bunch of bits they want. There are no proprietary or reserved bit patterns that processes cannot create. Also, if a process has one copy of a particular set of bits, it's trivial to create more copies of it. The first characteristic implies that it's possible for anyone at all to create any capability they want. The second characteristic implies that once someone has a working capability, they can make as many copies of it as they want, and can potentially store them anywhere they want, including on an entirely different machine.

That doesn't sound so good from a security perspective. If a process needs a capability with a particular bit pattern to open `/tmp/foo` for read and write, maybe it can just generate that bit pattern and successfully give itself the desired access to the file. That's not what we're looking for in an access control mechanism. We want capabilities to be unforgeable. Even if we can get around that problem, the ability to copy a capability would suggest we can't take access permission away, once granted, since the process might have copies of the capability stashed away elsewhere[6]. Further, perhaps the process can grant access to another process merely by using IPC to transfer a copy of the capability to that other process.

We typically deal with these issues when using capabilities for access control by never letting a process get its metaphoric hands on any capability. The operating system controls and maintains capabilities, storing them somewhere in its protected memory space. Processes can perform various operations on capabilities, but only with the mediation of the operating system. If, for example, process A wishes to give process B read/write access to file `/tmp/foo` using capabilities, A can't merely

[6] This ability is commonly called **revocation**. Revocation is easy with ACLs, since you just go to the ACL and change it. Depending on implementation, it can be easy or hard for capabilities.

send B the appropriate bit pattern. Instead, A must make a system call requesting the operating system to give the appropriate capability to B. That gives the OS a chance to decide whether its security policy permits B to access `/tmp/foo` and deny the capability transfer if it does not.

So if we want to rely on capabilities for access control, the operating system will need to maintain its own protected capability list for each process. That's simple enough, since the OS already has a per-process protected data structure, the PCB. Slap a pointer to the capability list (stored in kernel memory) into the process' PCB and you're all set. Now when the process attempts to open `/tmp/foo` for read/write, the call traps to the OS, the OS consults the capability list for that process to see if there is a relevant capability for the operation on the list and proceeds accordingly.

In a general system, keeping an on-line capability list of literally everything some principal is permitted to access would incur high overheads. If we used capabilities for file-based access control, a user might have thousands of capabilities, one for each file the user was allowed to access in any way. Generally, if one is using capabilities, the system persistently stores the capabilities somewhere safe, and imports them as needed. So a capability list attached to a process is not necessarily very long, but there is an issue of deciding which capabilities of the immense set users have at their discretion to give to each process they run.

There is another option. Capabilities need not be stored in the operating system. Instead, they can be cryptographically protected. If capabilities are relatively long and are created with strong cryptography, they cannot be guessed in a practical way and can be left in the user's hands. Cryptographic capabilities make most sense in a distributed system, so we'll talk about them in the chapter on distributed system security.

There are good and bad points about capabilities, just as there were for access control lists. With capabilities, it's easy to determine which system resources a given principal can access. Just look through the principal's capability list. Revoking access merely requires removing the capability from the list, which is easy enough if the OS has exclusive access to the capability (but much more difficult if it does not). If you have the capability readily available in memory, it can be quite cheap to check it, particularly since the capability can itself contain a pointer to the data or software associated with the resource it protects. Perhaps merely having such a pointer is the system's core implementation of capabilities.

On the other hand, determining the entire set of principals who can access a resource becomes more expensive. Any principal might have a capability for the resource, so you must check all principals' capability lists to tell. Simple methods for making capability lists short and manageable have not been as well developed as the Unix method of providing short ACLs. Also, the system must be able to create, store, and retrieve capabilities in a way that overcomes the forgery problem, which can be challenging.

One neat aspect of capabilities is that they offer a good way to create processes with limited privileges. With access control lists, a process in-

herits the identity of its parent process, also inheriting all of the privileges of that principal. It's hard to give the process just a subset of the parent's privileges. Either you need to create a new principal with those limited privileges, change a bunch of access control lists, and set the new process's identity to that new principal, or you need some extension to your access control model that doesn't behave quite the way access control lists ordinarily do. With capabilities, it's easy. If the parent has capabilities for X, Y, and Z, but only wants the child process to have the X and Y capabilities, when the child is created, the parent transfers X and Y, not Z.

In practice, user-visible access control mechanisms tend to use access control lists, not capabilities, for a number of reasons. However, under the covers operating systems make extensive use of capabilities. For example, in a typical Linux system, that `open()` call we were discussing uses ACLs for access control. However, assuming the Linux `open()` was successful, as long as the process keeps the file open, the ACL is not examined on subsequent reads and writes. Instead, Linux creates a data structure that amounts to a capability indicating that the process has read and write privileges for that file. This structure is attached to the process's PCB. On each read or write operation, the OS can simply consult this data structure to determine if reading and writing are allowed, without having to find the file's access control list. If the file is closed, this capability-like structure is deleted from the PCB and the process can no longer access the file without performing another `open()` which goes back to the ACL. Similar techniques can be used to control access to hardware devices and IPC channels, especially since UNIX-like systems treat these resources as if they were files. This combined use of ACLs and capabilities allows the system to avoid some of the problems associated with each mechanism. The cost of checking an access control list on every operation is saved because this form of capability is easy to check, being merely the presence or absence of a pointer in an operating system data structure. The cost of managing capabilities for all accessible objects is avoided because the capability is only set up after a successful ACL check. If the object is never accessed by a process, the ACL is never checked and no capability is required. Since any given process typically opens only a tiny fraction of all the files it is permitted to open, the scaling issue doesn't usually arise.

55.5 Mandatory And Discretionary Access Control

Who gets to decide what the access control on a computer resource should be? For most people, the answer seems obvious: whoever owns the resource. In the case of a user's file, the user should determine access control settings. In the case of a system resource, the system administrator, or perhaps the owner of the computer, should determine them. However, for some systems and some security policies, that's not the right answer. In particular, the parties who care most about information security sometimes want tighter controls than that.

The military is the most obvious example. We've all heard of Top Secret information, and probably all understand that even if you are allowed to see Top Secret information, you're not supposed to let other people see it, too. And that's true even if the information in question is in a file that you created yourself, such as a report that contains statistics or quotations from some other Top Secret document. In these cases, the simple answer of the creator controlling access permissions isn't right. Whoever is in overall charge of information security in the organization needs to make those decisions, which implies that principal has the power to set the access controls for information created by and belonging to other users, and that those users can't override his decisions. The more common case is called **discretionary access control**. Whether almost anyone or almost no one is given access to a resource is at the discretion of the owning user. The more restrictive case is called **mandatory access control**. At least some elements of the access control decisions in such systems are mandated by an authority, who can override the desires of the owner of the information. The choice of discretionary or mandatory access control is orthogonal to whether you use ACLs or capabilities, and is often independent of other aspects of the access control mechanism, such as how access information is stored and handled. A mandatory access control system can also include discretionary elements, which allow further restriction (but not loosening) of mandatory controls.

Many people will never work with a system running mandatory access controls, so we won't go further into how they work, beyond observing that clearly the operating system is going to be involved in enforcing them. Should you ever need to work in an environment where mandatory access control is important, you can be sure you will hear about it. You should learn more about it at that point, since when someone cares enough to use mandatory access control mechanisms, they also care enough to punish users who don't follow the rules. Loscocco [L01] describes a special version of Linux that incorporates mandatory access control. This is a good paper to start with if you want to learn more about the characteristics of such systems.

55.6 Practicalities Of Access Control Mechanisms

Most systems expose either a simple or more powerful access control list mechanism to their users, and most of them use discretionary access control. However, given that a modern computer can easily have hundreds of thousands, or even millions of files, having human users individually set access control permissions on them is infeasible. Generally, the system allows each user to establish a default access permission that is used for every file he creates. If one uses the Linux `open()` call to create a file, one can specify which access permissions to initially assign to that file. Access permissions on newly created files in Unix/Linux systems can be further controlled by the **`umask()`** call, which applies to all new file creations by the process that performed it.

ASIDE: THE ANDROID ACCESS CONTROL MODEL

The Android system is one of the leading software platforms for today's mobile computing devices, especially smart phones. These devices pose different access control challenges than classic server computers, or even personal desktop computers or laptops. Their functionality is based on the use of many relatively small independent applications, commonly called apps, that are downloaded, installed, and run on a device belonging to only a single user. Thus, there is no issue of protecting multiple users on one machine from each other. If one used a standard access control model, these apps would run under that user's identity. But apps are developed by many entities, and some may be malicious. Further, most apps have no legitimate need for most of the resources on the device. If they are granted too many privileges, a malicious app can access the phone owner's contacts, make phone calls, or buy things over the network, among many other undesirable behaviors. The principle of least privilege implies that we should not give apps the full privileges belonging to owner, but they must have some privileges if they are to do anything interesting.

Android runs on top of a version of Linux, and an application's access limitations are achieved in part by generating a new user ID for each installed app. The app runs under that ID and its accesses can be controlled on that basis. However, the Android middleware offers additional facilities for controlling access. Application developers define accesses required by their app. When a user considers installing an app on their device, they are shown what permissions it requires. The user can either grant the app those permissions, not install the app, or limit its permissions, though the latter choice may also limit app utility. Also, the developer specifies ways in which other apps can communicate with the new app. The data structure used to encode this access information is called a **permission label**. An app's permission labels (both what it can access and what it provides to others) are set at app design time, and encoded into a particular Android system at the moment the app is installed on that machine.

Permission labels are thus like capabilities, since possession of them by the app allows the app to do something, while lacking a label prevents the app from doing that thing. An app's set of permission labels is set statically at install time. The user can subsequently change those permissions, although limiting them may damage app functionality. Permission labels are a form of mandatory access control. The Android security model is discussed in detail by Enck et al. [E+09].

The Android security approach is interesting, but not perfect. In particular, users are not always aware of the implications of granting an application access to something, and, faced with the choice of granting the access or not being able to effectively use the app, they will often grant it. This behavior can be problematic, if the app is malicious.

If desired, the owner can alter that initial ACL, but experience shows that users rarely do. This tendency demonstrates the importance of properly chosen defaults. Here, as in many other places in an operating system, a theoretically changeable or tunable setting will, in practice, be used unaltered by almost everyone almost always.

However, while many will never touch access controls on their resources, for an important set of users and systems these controls are of vital importance to achieve their security goals. Even if you mostly rely on defaults, many software installation packages use some degree of care in setting access controls on executables and configuration files they create. Generally, you should exercise caution in fiddling around with access controls in your system. If you don't know what you're doing, you might expose sensitive information or allow attackers to alter critical system settings. If you tighten existing access controls, you might suddenly cause a bunch of daemon programs running in the background to stop working.

One practical issue that many large institutions discovered when trying to use standard access control methods to implement their security policies is that people performing different roles within the organization require different privileges. For example, in a hospital, all doctors might have a set of privileges not given to all pharmacists, who themselves have privileges not given to the doctors. Organizing access control on the basis of such roles and then assigning particular users to the roles they are allowed to perform makes implementation of many security policies easier. This approach is particularly valuable if certain users are permitted to switch roles depending on the task they are currently performing, since then one need not worry about setting or changing the individual's access permissions on the fly, but simply switch their role from one to another. Usually they will hold the role's permission only as long as they maintain that role. Once they exit the particular role (perhaps to enter a different role with different privileges), they lose the privileges of the role they exit.

This observation led to the development of **Role-Based Access Control**, or **RBAC**. The core ideas had been around for some time before they were more formally laid out in a research paper by Ferraiolo and Kuhn [FK92]. Now RBAC is in common use in many organizations, particularly large ones. Large organizations face more serious management challenges than small ones, so approaches like RBAC that allow groups of users to be dealt with in one operation can significantly ease the management task. For example, if a company determines that all programmers should be granted access to a new library that has been developed, but accountants should not, RBAC would achieve this effect with a single operation that assigns the necessary privilege to the *Programmer* role. If a programmer is promoted to a management position for which access to the library is unnecessary, the company can merely remove the *Programmer* role from the set of roles the manager could take on.

Such restrictions do not necessarily imply that you suspect your accountants of being dishonest and prone to selling your secret library code to competitors[7]. Remember the principle of least privilege: when you give someone access to something, you are relying not just on their honesty, but on their caution. If accountants can't access the library at all,

[7] Dishonest accountants are generally good to avoid, so you probably did your best to hire honest ones, after all. Unless you're Bernie Madoff [W20], perhaps...

then neither malice nor carelessness on their part can lead to an accountant's privileges leaking your library code. Least privilege is not just a theoretically good idea, but a vital part of building secure systems in the real world.

RBAC sounds a bit like using groups in access control lists, and there is some similarity, but RBAC systems are a good deal more powerful than mere group access permissions; RBAC systems allow a particular user to take on multiple disjoint roles. Perhaps our programmer was promoted to a management position, but still needs access to the library, for example when another team member's code needs to be tested. An RBAC system would allow our programmer to switch between the role of manager and programmer, temporarily leaving behind rights associated with the manager and gaining rights associated with the programmer role. When the manager tested someone else's new code, the manager would have permission to access the library, but would *not* have permission to access team member performance reviews. Thus, if a sneaky programmer slipped malicious code into the library (e.g., that tried to read other team members' performance reviews, or learn their salaries), the manager running that code would not unintentionally leak that information; using the proper role at the proper time prevents it.

These systems often require a new authentication step to take on an RBAC role, and usually taking on Role A requires relinquishing privileges associated with one's previous role, say Role B. The manager's switch to the code testing role would result in temporarily relinquishing privileges to examine the performance reviews. On completing the testing, the manager would switch back to the role allowing access to the reviews, losing privilege to access the library. RBAC systems may also offer finer granularity than merely being able to read or write a file. A particular role (*Salesperson*, for instance) might be permitted to add a purchase record for a particular product to a file, but would not be permitted to add a re-stocking record for the same product to the same file, since salespeople don't do re-stocking. This degree of control is sometimes called **type enforcement**. It associates detailed access rules to particular objects using what is commonly called a **security context** for that object. How exactly this is done has implications for performance, storage of the security context information, and authentication.

One can build a very minimal RBAC system under Linux and similar OSes using ACLs and groups. These systems have a feature in their access control mechanism called **privilege escalation**. Privilege escalation allows careful extension of privileges, typically by allowing a particular program to run with a set of privileges beyond those of the user who invokes them. In Unix and Linux systems, this feature is called **`setuid`**, and it allows a program to run with privileges associated with a different user, generally a user who has privileges not normally available to the user who runs the program. However, those privileges are only granted during the run of that program and are lost when the program exits. A carefully written `setuid` program will only perform a limited set of oper-

TIP: PRIVILEGE ESCALATION CONSIDERED DANGEROUS

We just finished talking about how we could use privilege escalation to temporarily change what one of our users can do, and how this offers us new security options. But there's a dangerous side to privilege escalation. An attacker who breaks into your system frequently compromises a program running under an identity with very limited privileges. Perhaps all it's supposed to be able to do is work with a few simple informational files and provide remote users with their content, and maybe run standard utilities on those files. It might not even have write access to its files. You might think that this type of compromise has done little harm to the system, since the attacker cannot use the access to do very much.

This is where the danger of privilege escalation comes into play. Attackers who have gained any kind of a foothold on a system will then look around for ways to escalate their privileges. Even a fairly unprivileged application can do a lot of things that an outsider cannot directly do, so attackers look for flaws in the code or configuration that the compromised application can access. Such attempts to escalate privilege are usually an attacker's first order of business upon successful compromise of a system.

In many systems, there is a special user, often called the **superuser** or **root** user. This user has a lot more privilege than any other user on the system, since its purpose is to allow for the most vital and far-reaching system administration changes on that system. The paramount goal of an attacker with a foothold on your system is to use privilege escalation to become the root user. An attacker who can do that will effectively have total control of your system. Such an attacker can look at any file, alter any program, change any configuration, and perhaps even install a different operating system. This danger should point out how critical it is to be careful in allowing any path that permits privilege escalation up to superuser privilege.

ations using those privileges, ensuring that privileges cannot be abused[8]. One could create a simple RBAC system by defining an artificial user for each role and associating desired privileges with that user. Programs using those privileges could be designated as `setuid` to that user.

The Linux `sudo` command, which we encountered in the authentication chapter, offers this kind of functionality, allowing some designated users to run certain programs under another identity. For example,

```
sudo -u Programmer install newprogram
```

would run this install command under the identity of user `Programmer`, rather than the identity of the user who ran the command, assuming that user was on a system-maintained list of users allowed to take on the identity `Programmer`. Secure use of this approach requires careful configura-

[8] Unfortunately, not all programs run with the `setuid` feature are carefully written, which has led to many security problems over the years. Perhaps true for all security features, alas?

tion of system files controlling who is allowed to execute which programs under which identities. Usually the `sudo` command requires a new authentication step, as with other RBAC systems.

For more advanced purposes, RBAC systems typically support finer granularity and more careful tracking of role assignment than `setuid` and `sudo` operations allow. Such an RBAC system might be part of the operating system or might be some form of add-on to the system, or perhaps a programming environment. Often, if you're using RBAC, you also run some degree of mandatory access control. If not, in the example of `sudo` above, the user running under the `Programmer` identity could run a command to change the access permissions on files, making the `install` command available to non- programmers. With mandatory access control, a user could take on the role of `Programmer` to do the installation, but could not use that role to allow salespeople or accountants to perform the installation.

55.7 Summary

Implementing most security policies requires controlling which users can access which resources in which ways. Access control mechanisms built in to the operating system provide the necessary functionality. A good access control mechanism will provide complete mediation (or close to it) of security-relevant accesses through use of a carefully designed and implemented reference monitor.

Access control lists and capabilities are the two fundamental mechanisms used by most access control systems. Access control lists specify precisely which subjects can access which objects in which ways. Presence or absence on the relevant list determines if access is granted. Capabilities work more like keys in a lock. Possession of the correct capability is sufficient proof that access to a resource should be permitted. User-visible access control is more commonly achieved with a form of access control list, but capabilities are often built in to the operating system at a level below what the user sees. Neither of these access control mechanisms is inherently better or worse than the other. Rather, like so many options in system design, they have properties that are well suited to some situations and uses and poorly suited to others. You need to understand how to choose which one to use in which circumstance.

Access control mechanisms can be discretionary or mandatory. Some systems include both. Enhancements like type enforcement and role-based access control can make it easier to achieve the security policy you require.

Even if the access control mechanism is completely correct and extremely efficient, it can do no more than implement the security policies that it is given. Security failures due to faulty access control mechanisms are rare. Security failures due to poorly designed policies implemented by those mechanisms are not.

References

[C+63] "The Compatible Time Sharing System: A Programmer's Guide" by F. J. Corbato, M. M. Daggett, R. C. Daley, R. J. Creasy, J. D. Hellwig, R. H. Orenstein, and L. K. Korn. M.I.T. Press, 1963. *The programmer's guide for the early and influential CTSS time sharing system. Referenced here because it used an early version of an access control list approach to protecting data stored on disk.*

[DV64] "Programming Semantics for Multiprogrammed Computations" by Jack B. Dennis and Earl. C. van Horn. Communications of the ACM, Vol. 9, No. 3, March 1966. *The earliest discussion of the use of capabilities to perform access control in a computer. Though the authors themselves point to the "program reference table" used in the Burroughs B5000 system as an inspiration for this notion.*

[E+09] "Understanding Android Security" by William Enck, Machigar Ongtang, and Patrick McDaniel. IEEE Security and Privacy, Vol. 7, No. 1, January/February 1999. *An interesting approach to providing access control in a particular and important kind of machine. The approach has not been uniformly successful, but it is worth understanding in more detail than we discuss in this chapter.*

[FK92] "Role-Based Access Controls" by David Ferraiolo and D. Richard Kuhn. 15th National Computer Security Conference, October 1992. *The concepts behind RBAC were floating around since at least the 70s, but this paper is commonly regarded as the first discussion of RBAC as a formal concept with particular properties.*

[L84] "Capability-Based Computer Systems" by Henry Levy. Digital Press, 1984. *A full book on the use of capabilities in computer systems, as of 1984. It includes coverage of both hardware using capabilities and operating systems, like Hydra, that used them.*

[L01] "Integrating Flexible Support for Security Policies Into the Linux Operating System" by Peter Loscocco. Proceedings of the FREENIX Track at the USENIX Annual Technical Conference 2001. *The NSA built this version of Linux that incorporates mandatory access control and other security features into Linux. A good place to dive into the world of mandatory access control, if either necessity or interest motivates you to do so.*

[S74] "Protection and Control of Information Sharing in Multics" by Jerome Saltzer. Communications of the ACM, Vol. 17, No. 7, July 1974. *Sometimes it seems that every system idea not introduced in CTSS was added in Multics. In this case, it's the general use of groups in access control lists.*

[T84] "Reflections on Trusting Trust" by Ken Thompson. Communications of the ACM, Vol. 27, No. 8, August 1984. *Ken Thompson's Turing Award lecture, in which he pointed out how sly systems developers can slip in backdoors without anyone being aware of it. People have wondered ever since if he actually did what he talked about...*

[W20] "Bernie Madoff" by Wikipedia. `https://en.wikipedia.org/wiki/Bernie_Madoff`. *Bernie Madoff (painfully, pronounced "made off", as in "made off with your money") built a sophisticated Ponzi scheme, a fraud of unimaginable proportions (nearly 100 billion dollars). He is, as Wikipedia says, an "American charlatan". As relevant here, he probably hired dishonest accountants, or was one himself.*

[W+74] "Hydra: The Kernel of a Multiprocessor Operating System" by W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pearson, and F. Pollack. Communications of the ACM, Vol. 17, No. 6, June 1974. *A paper on a well-known operating system that made extensive and sophisticated use of capabilities to handle access control.*