Security for Operating Systems

Introduction

Security of computing systems is a vital topic whose importance only keeps increasing. Much money has been lost and many people's lives have been harmed when computer security has failed. Attacks on computer systems are so common as to be inevitable in almost any scenario where you perform computing. Generally, all elements of a computer system can be subject to attack, and flaws in any of them can give an attacker an opportunity to do something you want to prevent. But operating systems are particularly important from a security perspective. Why?

To begin with, pretty much everything runs on top of an operating system. As a rule, if the software you are running on top of, whether it be an operating system, a piece of middleware, or something else, is insecure, what's above it is going to also be insecure. It's like building a house on sand. You may build a nice solid structure, but a flood can still wash away the base underneath your home, totally destroying it despite the care you took in its construction. Similarly, your application might perhaps have no security flaws of its own, but if the attacker can misuse the software underneath you to steal your information, crash your program, or otherwise cause you harm, your own efforts to secure your code might be for naught.

This point is especially important for operating systems. You might not care about the security of a particular web server or database system if you don't run that software, and you might not care about the security of some middleware platform that you don't use, but everyone runs an operating system. Thus, security flaws in an operating system, especially a widely used one, have an immense impact on many users and many pieces of software.

Another reason that operating system security is so important is that ultimately all of our software relies on proper behavior of the underlying hardware: the processor, the memory, and the peripheral devices. What has ultimate control of those hardware resources? The operating system. It can make your memory readable or writeable by another process, it can starve you of cycles, it can corrupt the messages you send via your network card, it can wipe data off your disk, or do pretty much anything else with the hardware you rely on. You trust the operating system not to do any of those things, but your trust relies on the assumption that the OS contains no flaws that will allow an attacker to make it misbehave.

Thinking about what you have already studied concerning memory management, scheduling, file systems, synchronization, and so forth, what would happen with each of these components of your operating system if an opponent could force it to behave in some arbitrarily bad way? If you understand what you've learned so far, you should find this prospect deeply disturbing. Our computing lives depend on our operating systems behaving as they have been defined to behave, and particularly on them not behaving in ways that benefit our opponents, rather than us.

The task of securing an operating system is not an easy one, since modern operating systems are large and complex. Your experience in writing code should have already pointed out to you that the more code you've got, and the more complex the algorithms are, the more likely your code is to contain flaws. Failures in software security generally arise from these kinds of flaws. So large, complex programs are likely to be harder to secure than small, simple programs. Not many other programs are as large and complex as a modern operating system.

Another challenge in securing operating systems is that they are, for the most part, meant to support multiple processes simultaneously. As you've learned, there are many mechanisms in an operating system meant to segregate processes from each other, and to protect shared pieces of hardware from being used in ways that interfere with other processes. If we fully trusted every process running on our machine to do anything it wants with any hardware resource and any piece of data on the machine, securing the system would be a lot easier. However, we typically don't trust everything equally. When you download and run a script from a web site you haven't visited before, do you really want it to be able to wipe every file from your disk, kill all your other processes, and start using your network interface to send spam email to other machines? Probably not, but if you are the owner of your computer, you have the right to do all those things, if that's what you want to do. And unless the operating system is careful, any process it runs, including the one running that script you downloaded, can do anything you can do.

Consider the issue of operating system security from a different perspective. One role of an operating system is to provide useful abstractions for application programs to build on. These applications must rely on the OS implementations of the abstractions to work as they are defined. Often, one part of the definition of such abstractions is their security behavior. For example, we expect that the operating system's file system will enforce the access restrictions it is supposed to enforce. Applications can then build on this expectation to achieve the security goals they require, such as counting on the file system access guarantees to ensure that a file they have specified as unwritable does not get altered. If the applications cannot rely on proper implementation of OS abstraction security guarantees, then they cannot use these abstractions to achieve their own security goals. At the minimum, that implies a great deal more work on the part of the application developers, since they will need to take extra measures to achieve their desired security goals. Taking into account our earlier discussion, they will often be unable to achieve these goals if the abstractions they must rely on (such as virtual memory and a well-defined scheduling policy) cannot be trusted.

THE CRUX OF THE PROBLEM
HOW CAN WE SECURE OS RESOURCES?

In the face of multiple possibly concurrent and interacting processes running on the same machine, how can we ensure that the resources each process is permitted to access are exactly those it should access, in exactly the ways we desire? What primitives are needed from the OS? What mechanisms should be provided by the hardware? How can we use them to solve the problems of security?

Obviously operating system security is vital, yet hard to achieve. So what do we do to secure our operating system? Addressing that question has been a challenge for generations of computer scientists, and there is as yet no complete answer. But there are some important principles and tools we can use to secure operating systems. These are generally built into any general-purpose operating system you are likely to work with, and they alter what can be done with that system and how you go about doing it. So even if you don't personally care at all about security, understanding what your OS does to secure itself is necessary to understand how to get the system to do what you want.

What Are We Protecting?

We aren't likely to achieve good protection unless we have a fairly comprehensive view of what we're trying to protect when we say our operating system should be secure. Fortunately, that question is easy to answer for an operating system, at least at the high level: everything. That answer isn't very comforting, but it is best to have a realistic understanding of the broad implications of operating system security.

A typical commodity operating system has complete control of all hardware on the machine and is able to do literally anything the hardware permits. That means it can control the processor, read and write all registers, examine any main memory location, and perform any operation one of its peripherals supports. As a result, among the things the OS can do are:

- examine or alter any process' memory
- read, write, delete or corrupt any file on any writeable persistent storage medium, including hard disks and flash drives
- change the scheduling or even halt execution of any process
- send any message to anywhere, including altered versions of those a process wished to send
- enable or disable any peripheral device
- give any process access to any other process' resources
- arbitrarily take away any resource a process controls
- respond to any system call with a maximally harmful lie

In essence, processes are at the mercy of the operating system. It is nearly impossible for a process to "protect" any part of itself from a malicious operating system. We typically assume our operating system is not actually malicious[1], but a flaw that allows a malicious process to cause the operating system to misbehave is nearly as bad, since it could potentially allow that process to gain any of the powers of the operating system itself.

This point should make you think very seriously about the importance of designing secure operating systems and, more commonly, applying security patches to any operating system you are running. Security flaws in your operating system can

[1] If you suspect your operating system is malicious, it's time to get a new operating system.

completely compromise everything about the machine the system runs on, so preventing them and patching any that are found is vitally important.

Security Goals and Policies

What do we mean when we say we want an operating system, or any system, to be secure? That's a rather vague statement. What we really mean is that there are things we would like to happen in the system and things we don't want to happen, and we'd like a high degree of assurance that we get what we want. As in most other aspects of life, we usually end up paying for what we get, so it's worthwhile to think about exactly what security properties and effects we actually need and then pay only for those, not for other things we don't need. What this boils down to is that we want to specify the goals we have for the security-relevant behavior of our system and choose defense approaches likely to achieve those goals at a reasonable cost.

Researchers in security have thought about this issue in broad terms for a long time. At a high conceptual level, they have defined three big security-related goals that are common to many systems, including operating systems. They are:

- *Confidentiality* – Keep your secrets. If some piece of information is supposed to be hidden from others, don't allow them to find it out. For example, you don't want someone to learn what your credit card number is – you want that number kept confidential.
- *Integrity* – If some piece of information or component of a system is supposed to be in a particular state, don't allow an opponent to change it. For example, if you've placed an online order for delivery of one pepperoni pizza, you don't want a malicious prankster to change your order to 1000 anchovy pizzas.
- *Availability* – If some information or service is supposed to be available for your own or others' use, make sure an attacker cannot prevent its use. For example, if your business is having a big sale, you don't want your competitors to be able to block off the streets around your store, preventing your customers from reaching you.

These are big, general goals. For a real system, you need to drill down to more detailed, specific goals. In a typical operating system, for example, we might have a confidentiality goal stating that a process' memory space cannot be arbitrarily read by another process. We might have an integrity goal stating that if a user writes a record to a particular file, another user who should not be able to write that file can't change the record. We might have an availability goal stating that one process running on the system cannot hog the CPU and prevent other processes from getting their share of the CPU. If you think back on what you've learned about the process abstraction, memory management, scheduling, file systems, IPC, and other topics from this class, you should be able to think of some other obvious confidentiality, integrity, and availability goals we are likely to want in our operating systems.

For any particular system, even goals at this level are not sufficiently specific. The integrity goal alluded to above, where a user's file should not be overwritten by another user not permitted to do so, gives you a hint about the extra specificity we need in our security goals for a particular system. Maybe there is some user who should be able to overwrite the file, as might be the case when two people are collaborating on writing a report. But that doesn't mean an unrelated third user should be able to write that file, if he is not collaborating on the report stored there. We need to be able to specify such detail in our security goals. Operating systems are written to be used by many different people with many different needs, and operating system security should reflect that generality. What we want in security mechanisms for operating systems is flexibility in describing our detailed security goals.

ASIDE: **SECURITY VS. FAULT TOLERANCE**

When discussing the process abstraction, we talked about how virtualization protected a process from actions of other processes. For instance, we did not want our process' memory to be accidentally overwritten by another process, so our virtualization mechanisms had to prevent such behavior. Then we were talking primarily about flaws or mistakes in processes. Is this actually any different than worrying about malicious behavior, which is more commonly the context in which we discuss security? Have we already solved all our problems by virtualizing our resources?

Yes and no. (Isn't that a helpful phrase?) Yes, if we perfectly virtualized everything and allowed no interactions between anything, we very likely would have solved most problems of malice. However, most virtualization mechanisms are not totally bulletproof. They work well when no one tries to subvert them, but may not be perfect against all possible forms of misbehavior. Second, and perhaps more important, we don't totally isolate processes from each other. Processes share some OS resources by default (such as file systems) and can optionally choose to share others. These intentional relaxations of virtualization are not problematic when used properly, but they are also potential channels for malicious attacks. Finally, the OS does not always have complete control of the hardware . . .

Ultimately, of course, the operating system software must do its best to enforce those flexible security goals, which implies we'll need to encode those goals in forms that software can understand. We typically must convert our vague understandings of our security goals into very specific security *policies*. For example, in the case of the file described above, we might want to specify a policy like "users A and B may write to file X, but no other user can write it." With that degree of specificity, backed by carefully designed and implemented mechanisms, we can hope to achieve our security goals[2].

Note an important implication for operating system security: in many cases, an operating system will have the mechanisms necessary to implement a desired security policy with a high degree of assurance in its proper application, but only if someone tells the operating

[2] Yet another example of the operating system using general mechanisms to implement flexible policies.

system precisely what that policy is. With some important exceptions (like maintaining a process' address space private unless specifically directed otherwise), the operating system merely supplies general mechanisms that can implement many specific policies. Without intelligent design of policies and careful application of the mechanisms, however, what the operating system can do may not be what your operating system will do.

Designing Secure Systems

Few of you will ever build your own operating system, nor even make serious changes to any existing operating system, but we expect many of you will build large software systems of some kind. Experience of many computer scientists with system design has shown that there are certain design principles that are helpful in building systems with security requirements. These principles were originally laid out by Jerome Saltzer and Michael Schroeder in an influential paper [SS75], though some of them come from earlier observations by others. While neither the original authors nor later commentators would claim that following them will guarantee that your system is secure, paying attention to them has proven to lead to more secure systems, while you ignore them at your own peril. We'll discuss them briefly here. If you are actually building a large software system, it would be worth your while to look up this paper (or more detailed commentaries on it) and study the concepts more carefully.

1. Economy of mechanism – This basically means keep your system as small and simple as possible. Simple systems have fewer bugs and it's easier to understand their behavior. If you don't understand your system's behavior, you're not likely to know if it achieves security goals.
2. Fail safe defaults – Default to security, not insecurity. If policies can be set to determine the behavior of a system, have the default for those policies be more secure, not less.
3. Complete mediation – This is a security term meaning that you should check if an action to be performed meets security policies every single time the action is taken[3].
4. Open design – Assume your opponent knows every detail of your design. If the system can achieve its security goals anyway, you're in good shape. This principle does not necessarily mean that you actually tell everyone all the details, but base your security on the assumption that the attacker has learned everything. He often has, in practice.
5. Separation of privilege – Require separate parties or credentials to perform critical actions. For example, two-factor authentication, where you use both a password and possession of a piece of hardware to determine identity, is more secure than using either one of those methods alone.

[3] This particular principle is often ignored in many systems, in favor of lower overhead or usability. An overriding characteristic of all engineering design is that you often must balance conflicting goals, as we saw earlier in the course, such as in the scheduling chapters. We'll say more about that in the context of security later.

6. Least privilege – Give a user or a process the minimum privileges required to perform the actions you wish to allow. The more privileges you give to a party, the greater the danger that they will abuse those privileges. Even if you are confident that the party is not malicious, if they make a mistake, an adversary can leverage their error to use their superfluous privileges in harmful ways.
7. Least common mechanism – For different users or processes, use separate data structures or mechanisms to handle them. For example, each process gets its own page table in a virtual memory system, ensuring that one process cannot access another's pages.
8. Acceptability – A critical property not dear to the hearts of many programmers. If your users won't use it, your system is worthless. Far too many promising secure systems have been abandoned because they asked too much of their users.

These are not the only useful pieces of advice on designing secure systems out there. There is also lots of good material on taking the next step, converting a good design into code that achieves the security you intended, and other material on how to evaluate whether the system you have built does indeed meet those goals. These issues are beyond the scope of this course, but are extremely important when the time comes for you to build large, complex systems. For discussion of approaches to secure programming, you might start with [SE13], if you are working in C. If you are working in another language, you should seek out a similar text specific to that language, since many secure coding problem are related to specifics of the language. For a comprehensive treatment on how to evaluate if your system is secure, start with [D+07]

The Basics of OS Security

In a typical operating system, then, we have some set of security goals, centered around various aspects of confidentiality, integrity, and availability. Some of these goals tend to be built in to the operating system model, while others are controlled by the owners or users of the system. The built-in goals are those that are extremely common, or must be ensured to make the more specific goals achievable. Most of these built-in goals relate to controlling process access to pieces of the hardware. That's because the hardware is shared by all the processes on a system, and unless the sharing is carefully controlled, one process can interfere with the security goals of another process. Other built-in goals relate to services that the operating system offers, such as file systems, memory management, and interprocess communications. If these services are not carefully controlled, processes can subvert the system's security goals.

Clearly, a lot of system security is going to be related to process handling. If the operating system can maintain a clean separation of processes that can only be broken with the operating system's help, then neither shared hardware nor operating system services can be used to subvert our security goals. That requirement implies that the operating system needs to be careful about allowing use of hardware and of its services. In many cases, the operating system has good opportunities to apply such caution. For example, the operating system controls virtual memory, which in turn completely controls which main memory addresses each process can access. Hardware support prevents a process from even naming a physical memory address that is not mapped into its virtual memory space.

System calls offer the operating system another opportunity to provide protection. In most operating systems, processes access system services by making an explicit system call, as was discussed in earlier chapters. As you have learned, system calls switch the execution mode from the processor's user mode to its supervisor mode, invoking an appropriate piece of operating system code as they do so. That code can determine which process made the system call and what service the process requested. Earlier, we only talked about how this could allow the operating system to call the proper piece of system code to perform the service, and to keep track of who to return control to when the service had been completed. But the same mechanism gives the operating system the opportunity to check if the requested service should be allowed under the system's security policy. Since access to peripheral devices is through device drivers, which are usually also accessed via system call, the same mechanism can ensure proper application of security policies for hardware access.

When a process performs a system call, then, the operating system will use the process identifier in the process control block or similar structure to determine the identity of the process. The OS can then use *access control mechanisms* to decide if the identified process is *authorized* to perform the requested action. If so, the OS either performs the action itself on behalf of the process or arranges for the process to perform it without further system intervention. If the process is not authorized, the OS can simply generate an error code for the system call and return control to the process, if the scheduling algorithm permits.

TIP: THE WEAKEST LINK

It's worthwhile to remember that the people attacking your systems share many characteristics with you. In particular, they're probably pretty smart and they probably are kind of lazy, in the positive sense that they don't do work that they don't need to do. That implies that attackers tend to go for the easiest possible way to overcome your system's security. They're not going to search for a zero-day buffer overflow if you've chosen "password" as your password to access the system.

The practical implication for you is that you should spend most of the time you devote to securing your system to identifying and strengthening your weakest link. Your weakest link is the least protected part of your system, the one that's easiest to attack, the one you can't hide away or augment with some external security system. Often, a running system's weakest link is actually its human users, not its software. You will have a hard time changing the behavior of people, but you can design the software bearing in mind that attackers may try to fool the legitimate users into misusing it. Remember that principle of least privilege? If an attacker can fool a user who has complete privileges into misusing the system, it will be a lot worse than fooling a user who can only damage his own assets.

Summary

The security of the operating system is vital for both its own and its applications' sakes. Security failures in this software allow essentially limitless bad consequences. While achieving system security is challenging, there are known design principles that can help. These principles are useful not only in designing operating systems, but in designing any large software system.

Achieving security in operating systems depends on the security goals one has. These goals will typically include goals related to confidentiality, integrity, and availability. In any given system, the more detailed particulars of these security goals vary, which implies that different systems will have different security policies intended to help them meet their specific security goals. As in other areas of operating system design, we handle these varying needs by separating the specific policies used by any particular system from the general mechanisms used to implement the policies by all systems.

The next question to address is, what mechanisms should our operating system contain to help us support general security policies? The virtualization of processes and memory is one helpful mechanism, since it allows us to control the behavior of processes to a large extent. We will describe several other useful operating system security mechanisms in the upcoming chapters.

References

[D+07] "The Art of Software Security Assessment"
Mark Dowd, John McDonald, and Justin Schuh
Addison-Wesley, 2007.
A long, comprehensive treatment of how to determine if your software system meets its security goals. It also contains useful advice on avoiding security problems in coding.

[SE13] "Secure Coding in C and C++"
Robert Seacord
Addison-Wesley, 2013.
A well regarded book on how to avoid major security mistakes in coding in C.

[SS75] "The Protection of Information in Computer Systems"
Jerome Saltzer and Michael Schroeder
Proceedings of the IEEE, Vol. 63, No. 9, September 1975.
A highly influential paper, particularly their codification of principles for secure system design.

Authentication for Operating Systems

Introduction

Given that we need to deal with a wide range of security goals and security policies that are meant to achieve those goals, what do we need from our operating system? Operating systems provide services for processes, and some of those services have security implications. Clearly, the operating system needs to be careful in such cases to do the right thing, security-wise. But the reason operating system services are allowed at all is that sometimes they need to be done, so any service that the operating system might be able to perform probably should be performed — under the right circumstances.

Context will be everything in operating system decisions on whether to perform some service or to refuse to do so because it will compromise security goals. Perhaps the most important element of that context is who's doing the asking. In the real world, if you significant other asks you to pick up a gallon of milk at the store on the way home, you'll probably do so, while if a stranger on the street asks the same thing, you probably won't. In an operating system context, if the system administrator asks the operating system to install a new program, it probably should, while if a script downloaded from a random web page asks to install a new program, the operating system should take more care before performing the installation.

So knowing who is requesting an operating system service is crucial in meeting your security goals. How does the operating system know that? Let's work a bit backwards here to figure it out.

Operating system services are most commonly requested by system calls, and system calls are made by particular processes. As discussed in the first part of this book, system calls result in a trap from user code into the operating system. The operating system then takes control and performs some service in response to the system call. At the moment the operating system takes control, it is able to determine which process made the call. Associated with that calling process is the OS-controlled data structure that describes the process, so the operating system can check that data structure to determine the identity of the process. Based on that identity, the operating system now has the opportunity to make a policy-based decision on whether to perform the requested operation.

In some cases, other information specific to the process is useful for making this decision. For instance, as we discovered when investigating virtualization of memory, per-process data structures ensure that only memory page frames assigned to a process can be accessed by that process. Also, when the operating system grants a process access to some resource, such as a file or a hardware device, an indication of that permission can be attached to the process control block, which will allow the operating system to determine, on later requests to use the resource, whether they should or shouldn't be permitted.

In other cases, however, the operating system needs more information about the process' identity to determine if its request should be granted. Often, some higher-level identity associated with a process is needed to make this decision. Different operating systems have used different types of identity for this purpose. For instance, most operating systems have a notion of a user identity, where the user is, typically, some human being. (The concept of a user has been expanded over the years to increase the power of the concept, as we'll see later.) So perhaps all processes run by a particular person will have the same identity associated with them. Another common type of identity is a group of users. In a manufacturing company, you might want to give all your salespersons access to your inventory information, so they can determine how many widgets and whizz-bangs you have in the warehouse, while it wouldn't be necessary for your human resources personnel to have access to that information[1]. Yet another form of identity is the program that the process is running. Recall that a process is a running version of a program. In some systems (such as the Android Operating System), you can grant certain privileges to particular programs. Whenever they run, they can use these privileges, but other programs cannot.

Regardless of the kind of identity we use to make our security decisions, we must have some way of attaching that identity to a particular process. Clearly, this attachment is a crucial security issue. If you misidentify a janitor employee process as an accounting department employee process, you could end up with an empty bank account. (And, since the janitor is fleeing to Rio with the loot, your trash cans won't get emptied.) Or if you fail to identify your company president correctly when he's trying to give an important presentation to investors, you may find yourself out of a job once he determines that you're the one who preventing him from getting the next round of startup capital. On the other hand, since everything except the operating system's own activities are performed by some process, if we can get this right for processes, we can be pretty sure we will have the opportunity to check our policy on every important action. Which leads us to the crux of the problem.

THE CRUX OF THE PROBLEM
HOW CAN WE SECURELY IDENTIFY PROCESSES?

For systems that have multiple identities associated with processes, how can we be sure that each process has the correct identity attached? As new processes are created, how can we be sure the new process has the correct identity? How can we be sure that malicious entities cannot change the identity of a process?

Attaching Identities to Processes

Where do processes come from? Usually they are created by other processes, though of course there's always that question of where the first process came from. One simple way to attach an identity to a new process, then, is simply to copy the identity of the

[1] Remember the principle of least privilege from the previous chapter? Here's an example of using it. A rogue human services employee won't be able to order your warehouse emptied of pop-doodles if you haven't given such employees the right to do so. As you read through the security chapters of this book, keep your eyes out for other applications of the security principles we discussed earlier.

process that created it. The child inherits the parent's identity. Mechanically, when the operating system services a call from old process A to create new process B (fork, for example), it consults A's process control block to determine A's identity, creates a new process control block for B, and copies in A's identity. Simple, no?

That's all well and good if all processes always have the same identity. We can create the primal process when our operating system boots, perhaps assigning it some special system identity not assigned to any human user. All other processes are its descendants and all of them inherit that single identity. But if there really is only one identity, what's the point in keeping track of it in the process control block? Since there's only one, you already know the identity. And, of course, if there's only one identity and we're going to use process identity to make security decisions about processes, we're not going to be able to implement any policy that distinguishes the handling of one process versus another.

We essentially have two choices on how to move forward: first, we can arrange that some processes have different identities and use that difference to manage our security policies. Or, second, we can use some other means to attach information to processes that will allow us to implement our policies. The second option immediately raises the question of how do we determine which information to attach to each process, which more or less leads us in a circle. Ultimately, if we want some processes to have different security policy treatment than others, we need, somehow or other, to identify which processes get which treatment. And that leads us back to process identity, in one way or another.

An easy escape from such circularity is simply to accept that there are different identities in the system and security treatment depends on determining which identity each process should have. A simple case is a multi-user system. We could then assign processes identities based on which human user they belong to. If our security policies are primarily about some people being allowed to do some things and others not being allowed to, we now have an idea of how we can go about making our decisions.

OK, so processes have a security-relevant identity, like a user ID. If not all processes belong to the same user ID, we're going to have to do something to set the proper user ID for a new process. In the ordinary case, a user has a process that he works with ordinarily: the shell process in command line systems, the window manager process in window-oriented system – you had figured out that both of these had to be processes themselves, right? So when you type a command into a shell or double click on an icon to start a process in a windowing system, you are asking the operating system to start a new process under your identity.

Great! But we do have another issue to deal with. How did that shell or window manager get your identity attached to itself? Here's where a little operating system privilege comes in handy. When a user first starts interacting with a system, the operating system can start a process up for him. Since the operating system can fiddle with its own data structures, like the process control block, it can set the new process' ownership to the user who just joined the system. Of course, we don't want users arbitrarily fiddling with the ownership of their processes themselves, but since the OS controls the process control block, the user can't alter the ownership information. Well, not without the OS's assistance, at least.

Again, well and good, but how did the operating system determine the user's identity so it could set process ownership properly? You probably can guess the answer – the user logged in, implying that the user provided identity information to the OS proving who he was. We've now identified a new requirement for the operating system: it must be able to query identity from human users and verify that they are who they claim to be, so we can attach reliable identities to processes, so we can use those identities to implement our security policies. One thing tends to lead to another in operating systems.

So how does the OS do that? As should be clear, we're sort of building a towering security structure with unforeseeable implications based on the OS making the right decision here, so it's very important. Let's look at our options.

How to Authenticate Users?

So this human being walks up to a computer . . .

Assuming we leave aside the possibilities for jokes, what can be done to allow the computer's system to determine who this person is, with reasonable accuracy? First, if the person is not an authorized user of the system at all, we should totally reject his attempt to sneak in. Second, if he is an authorized user, we need to determine, which one?

Classically, authenticating the identity of human beings has worked in one of three ways:

1. Authentication based on what you know
2. Authentication based on what you have
3. Authentication based on what you are

When we say "classically" here, we mean "classically" in the, well, classical sense. Classically as in going back to the ancient Greeks and Romans. For example, Polybius, writing in the second century B.C., describes how the Roman army used "watchwords" to distinguish friends from foes [p46], an example of authentication based on what you know. A Roman architect named Celer wrote a letter of recommendation (which still survives) for one of his slaves to be taken to an imperial procurator at some time in the 2nd century AD [C00]. Even further back, in (literally) Biblical times, the Gileadites required refugees after a battle to say the word "shibboleth," since the enemies they sought (the Ephraimites) could not properly pronounce that word [J]. This was a form of authentication by what you are: a native speaker of the Gileadites' dialect or a speaker of the Ephraimite dialect.

Having established the antiquity of these methods of authentication, let's leap past several centuries of history to the Computer Era to discuss how we use them in the context of computer authentication.

Authentication By What You Know

Authentication by what you know is most commonly performed by using passwords. Passwords have a long (and largely inglorious) history in computer security, going back at least to the CTSS system at MIT in the early 1960s [MT79]. A password is a secret known only to the party to be authenticated. By divulging the secret to the computer's operating system when attempting to log in, the party proves his identity. (You should be wondering about whether that implies that the system must also know the password, and what further implications that might have. We'll get to that.) The effectiveness of this form of authentication depends, obviously, on several factors. We're assuming other people don't know the party's password. If they do, the system gets fooled. We're assuming that no one else can guess it, either. And, of course, that the party in question does know (and remember) it.

Let's deal with the problem of other people knowing a password first. Leaving aside guessing, how could they know it? Someone who already knows it must let it slip, so the fewer parties who have to know it, the fewer parties we have to worry about. The person we're trying to authenticate has to know it, of course, since we're authenticating him based on him knowing it. We really don't want anyone else to be able to authenticate as that person to our system, so we'd prefer no third parties know the password. Thinking broadly about what a "third party" means here, that also implies the user shouldn't write the password down on a slip of paper, since anyone who steals the paper now knows the password. But there's one more party who would seem to need to know the password: our system itself. That suggests another possible vulnerability, since the system's copy of our password might leak out[2].

Actually, though, our system does not need to know the password. Think carefully about what the system is doing when it checks the password the user provides. It's checking to see if the user knows it, not to see what that password actually is. So if the user provides us the password, but we don't know the password, how on earth could our system do that?

You already know the answer, or at least you'll slap your forehead and say "I should have thought of that" once you hear it. Store a hash of the password, not the password itself. When the user provides you with what he claims to be the password, hash his claim and compare it to the stored hashed value. If it matches, you believe he knows the password. If it doesn't, you don't. Simple, no? And now your system doesn't need to store the actual password. That means if you're not too careful with how you store the authentication information, you haven't actually lost the passwords, just their hashes. By their nature, you can't reverse hashing algorithms, so the adversary can't use the stolen hash to obtain the password. There are some extra wrinkles here, but we'll leave those aside for the moment.

[2] "Might" is too weak a word. The first known incident of such stored passwords leaking is from 1962 [MT79], and such leaks happen to this day with depressing regularity, and general much larger scope. [KA16] discusses a 2016 leak of over 100 million passwords stored in plaintext form.

TIP: AVOID STORING SECRETS

Storing secrets like plaintext passwords or cryptographic keys is a hazardous business, since the secrets usually leak out. Protect your system by not storing them if you don't need to. If you do need to, store them in a hashed form using a strong cryptographic hash. If you can't do that, encrypt them with a secure cipher. (Perhaps you're complaining to yourself that we haven't told you about those yet. Be patient.) Store them in as few places, with as few copies as possible. Don't forget temporary editor files, backups, and the like, since the secrets may be there, too. Remember that anything you embed into an executable you give to others will not remain secret, so it's particularly dangerous to store secrets in executables.

Let's move on to the other problem: guessing. Can an attacker who wants to pose as a user simply guess the password? Consider the simplest possible password: a single bit, valued 0 or 1, like all other bits. If your password is a single bit long, then an attacker can try guessing "0" and have a 50/50 chances of being right. Even if he's wrong, if he can guess a second time, he now knows that the password is 1 and will correctly guess that.

Obviously, a one bit password is too easy to guess. How about an 8 bit password? Now there are 256 possible passwords you could choose. If the attacker guesses 256 times, he'll sooner or later guess right, taking 128 guesses, on average. Better than only having to guess twice, but still not good enough. It should be clear to you, at this point, that the length of the password is critical in being resistant to guessing. The longer the password, the harder to guess.

But there's another important factor, since we normally expect human beings to type in their passwords from keyboards or something similar. And given that we've already ruled out writing the password down somewhere as insecure, the person has to remember it. Early uses of passwords addressed this issue by restricting passwords to letters of the alphabet. While this made them easier to remember, it also cut down heavily on the number of bit patterns an attacker needed to guess to find someone's password, since all of the bit patterns that did not represent alphabetic characters would not appear in passwords. Over time, password systems have tended to expand the possible characters in a password, including upper and lower case letters, numbers, and special characters. The more possibilities, the harder to guess.

So we want long passwords composed of many different types of characters. But attackers know that people don't choose random strings of these types of characters as their passwords. They often choose names or familiar words, because those are easy to remember. Attackers trying to guess passwords will thus try lists of names and words before trying random strings of characters. This form of password guess is called a *dictionary attack*, and it can be highly effective. The dictionary here isn't Websters (or even the OED), but rather is a specialized list of words, names, meaningful strings of numbers (like "123456"), and other character patterns people tend to use for passwords, ordered by the probability that they will be chosen as the password. A good dictionary attack can figure out 90% of the passwords for a typical site [G13].

There are other troubling issues for the use of passwords, but many of those are not particular to operating systems, so we won't fling further mud at them here. Suffice it to say that there is a widely held belief in the computer security community that passwords are a technology of the past, and are no longer sufficiently secure for today's environments. At best, they can serve as one of several authentication mechanisms used in concert. This idea is called multi-factor authentication, with two-factor authentication being the version that gets the most publicity. You're perhaps already familiar with the concept: to get money out of an ATM, you need to know your personal identification number, or PIN. That's essentially a password. But you also need to provide further evidence of your identity . . .

Authentication by What You Have

Most of us have probably been in some situation where we had an identity card that we needed to show to get us into somewhere. At least, we've probably all attended some event where admission depended on having a ticket for the event. Those are both examples of authentication based on what you have, an ID card or a ticket, in these cases.

When authenticating yourself to an operating system, things are a bit different. In special cases, like the ATM mentioned above, the device (which is, after all, a computer inside – you knew that, right?) has special hardware to read our ATM card. That hardware allows it to determine that, yes, we have that card, thus providing the further proof to go along with your PIN. Most desktop computers, laptops, tablets, smart phones, and the like do not have that special hardware. So how can they tell what we have?

If we have something that plugs into one of the ports on a computer, such as a hardware token that uses USB, then, with suitable software support, the operating system can tell whether the user trying to log in has the proper device or not. Some security tokens are designed to work that way.

In other cases, since we're trying to authenticate a human user anyway, we make use of the person's capabilities to transfer information from whatever it is we have to the system we need to authenticate ourselves to. For example, some smart tokens display a number or character string on a tiny built-in screen. The human user types the information read off that screen into the computer's keyboard. The operating system does not get direct proof that the user has the device, but if only someone with access to the device could know what information he was supposed to type in, the evidence is nearly as good.

These kinds of devices rely on frequent changes of whatever information the device passes (directly or indirectly) to the operating system, perhaps every few seconds, perhaps every time the user tries to authenticate himself. Why? Well, if it doesn't, anyone who can learn the static information from the device no longer needs the device to pose as the user. The authentication mechanism has been converted from "something you have" to "something you know," and its security now depends on how hard it is for an attacker to learn that secret.

One weak point for all forms of authentication based on what you have is, what if you don't have it? What if you left it on your dresser bureau this morning? What if it slipped out of your pocket on your commute to work? What if a subtle pickpocket brushed up against you at the coffee shop and made off with it? You now have a two-fold problem. First, you don't have the magic item you need to authenticate yourself to the operating system. You can whine at your computer all you want, but it won't care. It will continue to insist that you produce the magic item you lost. Second, someone else has your magic item, and possibly they can pretend to be you, fooling the operating system that was relying on authentication by what you have. Note that the multi-factor authentication we mentioned earlier can save your bacon here, too. If the thief stole your security token, but doesn't know your password, he'll still have to guess that before he can pose as you.

If you study system security in practice for very long, you'll find that there's a significant gap between what academics (like me) tell you is safe and what happens in the real world. Part of this gap is because the real world needs to deal with real issues, like user convenience. Part of it is because security academics have a tendency to denigrate anything where they can think of a way to subvert it, even if that way is not itself particularly practical. One example in the realm of authentication mechanisms based on what you have is authenticating a user to a system by sending a text message to the user's cell phone. The user then types his message into the computer. Thinking about this in theory, this sounds very weak. In addition to the danger of losing the phone, security experts like to think about exotic attacks where the text message is misdirected to the attacker's phone, allowing him to provide the secret information from the text message to the computer.

In practice, people usually have their phone with them and take reasonable care not to lose it. If they do lose it, they notice that quickly and take equally quick action to fix their problem. So there is likely to be a relatively small window of time between when your phone is lost and when systems learn that they can't authenticate you using that phone. Also in practice, redirecting text messages sent to cell phones is possible, but far from trivial. The effort involved is likely to outweigh any benefit the attacker would get from fooling the authentication system, at least in the vast majority of cases. So a mechanism that causes security purists to avert their gazes in horror in actual use provides quite reasonable security[3]. Keep this lesson in mind. Even if it isn't on the test, it may come in handy some time in your later career.

Authentication by What You Are

If you don't like methods like passwords and you don't like having to hand out smart cards or security tokens to your users, there is another option. Human beings (who are what we're talking about authenticating here) are unique creatures. Each person has physical characteristics that differ from all others, sometimes in subtle ways, sometimes in obvious ones. In addition to properties of the human body (from DNA at the base up to the appearance of our face at the top), there are characteristics of human behavior that

[3] However, in 2016 the United States National Institute of Standards and Technology issued draft guidance deprecating the use of this technique for two-factor authentication, at least in some circumstances. Here's another security lesson: what works today might not work tomorrow.

are unique, or at least not shared by very many others. This observation suggests that if our operating system can only accurately measure these properties or characteristics, it can distinguish one person from another, solving our authentication problem.

This approach is very attractive to many people, most especially to those who have never tried to make it work. Going from the basic observation to a working, reliable authentication system is far from easy. But it can be made to work, to around the same extent that the other authentication mechanisms can be made to work. In other words, we can use it, but it is not likely to be perfect. And it has its own set of characteristic problems and challenges, different from those when you authenticate based on what you know or what you have, but not necessarily any easier to deal with.

Remember that we're talking about a computer program (either the OS itself or some separate program it invokes for the purpose) measuring a human characteristic and then determining if it belongs to a particular person or not. Spend a moment thinking about what that entails. To give you thinking some solidity, what if we plan to use facial recognition with the camera on a smart phone to authenticate the owner of the phone? If we decide it's the right person, we allow whoever we took the picture of to use the phone. If not, we give them the raspberry (in the cyber sense) and keep them out.

If you actually did think about it, like we told you to, you should have identified a few challenges here. Let's walk through things step by step, talking about some of those challenges. First, the camera is going to take a picture of someone who is, presumably, holding the phone. Maybe it's the owner, maybe it isn't. That's the point of taking the picture. If it isn't, we should assume whoever it is would like to fool us into thinking he's the actual owner. What if it's someone who looks a lot like the right user, but isn't? What if he's wearing a mask? What if he holds up a photo of the right user, instead of showing his own face? What if the lighting is dim, or he's not fully facing the camera? Alternately, what if it is the right user and he's not facing the camera, or the lighting is dim, or he just shaved off his beard?

Computer programs don't recognize faces the way people do. They do what programs always do with data: they convert it to zeros and ones and process it. So that "photo" you took is actually a collection of numbers, indicating shadow and light, shades of color, contrasts, and the like. So whoever you took a picture of, you've converted it to zeros and ones. OK, now what? Time to decide if it's the right person's photo or not! How is your program going to do that?

If it were a password, we could have stored the right password (or, better, a hash of the right password) and done a comparison of what got typed in (or its hash) to what we stored. If it's a perfect match, authenticate. Otherwise, don't. Can we do the same with this collection of zeros and ones that represent the picture we just took? Can we have a picture of the right user stored permanently in some file and compare the data from the camera to that file?

Probably not in the same way we compared the passwords. Consider one of those factors we just mentioned above: lighting. If the picture we stored in the file was taken under bright lights and the picture coming out of the camera was taken under dim lights, the two sets of zeros and ones are most certainly not going to match. In fact, it's quite unlikely that two pictures of the same person, taken a second apart under identical conditions, would be represented by exactly the same set of bits. So clearly we can't do a comparison based on bit-for-bit equivalence.

Instead, we need to compare based on a higher-level analysis of the two photos, the stored one of the right user and the just-taken one of the person who claims to be that user. There are a lot of ways we can do this, but generally they will involve trying to extract higher-level features from the photos and comparing those. We might, for example, try to calculate the length of the nose, or determine the color of the eyes, or make some kind of model of the shape of the mouth. Then we would compare the same feature set from the two photos.

ASIDE: SECURITY SNAKE OIL

If you spend much time worrying about security in a production environment, you'll run into a lot of folks who would like you to pay them for their security product, which invariably is described as tremendously improving the security of your environment. Some of these products are excellent and will indeed help you secure your systems. Others, however, are not any use at all. Like Old West medicine show barkers, they're just selling snake oil that won't cure anything.

Biometrics are one area where this phenomena arises. If you listen to the manufacturers, no one ever marketed a biometric authentication system that was less than perfect. Unfortunately, that's not always quite the truth. A good cautionary tale comes from Japanese researchers who obtained copies of 11 commercially available fingerprint readers, went to a hobby store and purchased around $10 worth of supplies, and tried to deceive the readers. They used the hobby supplies to make gummy fingers, copying fingerprints onto them in various ways. They were able to fool the fingerprint readers at rates between 68% and 100% using their gummy fingers [M+02].

The bigger the claims, the less likely they are to be true. The more "revolutionary" the technology, the more likely it is to be ineffective. To complicate life, that's not always true, but remember that an extraordinary claim requires extraordinary proof [TR78]

Even here, though, an exact match is not too likely. The lighting, for example, might slightly alter the perceived eye color: Paul-Newman blue eyes might become Henry-Fonda blue eyes under different lighting. So we'll need to allow some sloppiness in our comparison. If the feature match is "close enough," we authenticate. If not, we don't. We could go into this issue in a lot more detail, but the important point here is that when we are comparing facial features, or any other biometric, we cannot require perfect matches. Instead, we will look for close matches, which brings the nose of the camel of tolerances into our authentication tent. If we are intolerant of all but the closest matches, on some days we will fail to match the real user's picture to the stored version. (That's called a false negative, since we incorrectly decided not to authenticate.) If we are too tolerant of differences in the measured versus stored data, we will authenticate a user who is not who he claims to be. (That's called a false positive, since we incorrectly decided to authenticate.)

The nature of biometrics is that any implementation will have a characteristic false positive and false negative rate. Both are bad, so you'd like both to be low. For any given implementation of some biometric authentication technique, you can typically tune it to achieve some false positive rate, or tune it to achieve some false negative rate. But you can't usually minimize both. As the false positive rate goes down, the false negative rate goes up, and vice versa.

shows the typical relationship between these error rates. Note the circle at the point where the two curves cross. That point represents the *crossover error rate*, a common metric for describing the accuracy of a biometric. It represents an equal tradeoff between the two kinds of errors. It's not always the case that one tunes a biometric system to hit the crossover error rate, since you might care more about one kind of error than the other. For example, a smart phone that frequently locks its legitimate user out because it doesn't like today's fingerprint reading is not going to be popular, while the chances of a thief who stole the phone having a similar fingerprint are low. Perhaps low false negatives matter more here. On the other hand, if you're opening a bank vault with a retinal scan, once in a while requiring the bank manager to provide a scan a second time isn't too bad, while allowing a robber to open the vault with his bogus fake eye would be a disaster. Low false positives might be better here.

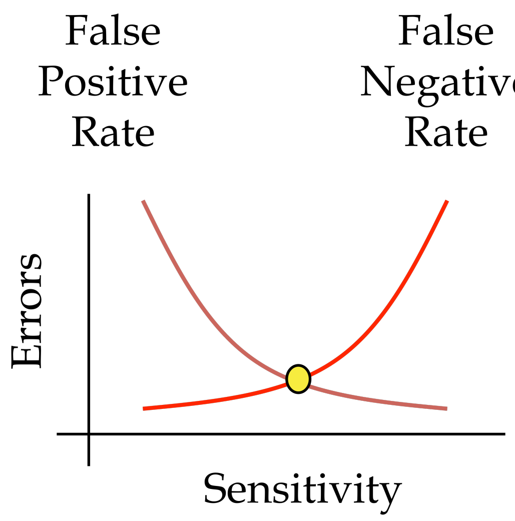

Figure 1. The Crossover Error Rate

Leaving aside the issues of reliability of authentication using biometrics, another big issue for using human characteristics to authenticate is that many of the techniques for measuring them require special hardware not likely to be present on most machines. Many computers (including smart phones, tablets, and laptops) are likely to have cameras, but embedded devices and server machines probably don't. Relatively few machines have fingerprint readers, and even fewer are able to measure more exotic biometrics. Even if the hardware device is available, the convenience of using them for this purpose can be limiting.

ASIDE: OTHER AUTHENTICATION POSSIBILITIES

Usually, what you know, what you have, and what you are cover the useful authentication possibilities, but sometimes there are other options. Consider going into the Department of Motor Vehicles to apply for a driver's license. You probably go up to a counter and talk to someone behind that counter, perhaps giving him a bunch of personal information, maybe even giving him some money to cover a fee for the license. Why on earth did you believe that person was actually a DMV employee who was able to get you a legitimate driver's license? You probably didn't know him, he didn't show you an official ID card, he didn't recite the secret DMV mantra that proved he was an initiate of that agency. You believed it because he was standing behind a particular counter, which is the counter DMV employees stand behind. You authenticated him based on where he was.

Once in a while, that approach can be handy in computer systems, most frequently in mobile or pervasive computing. If you're tempted to use it, think carefully about how you're obtaining the evidence that the subject really is in a particular place. It's actually fairly tricky.

What else? Perhaps you can sometimes authenticate based on what someone does. If you're looking for personally characteristic behavior, like their typing pattern or delays between commands, that's a type of biometric. But you might be less interested in authenticating exactly who they are versus authenticating that they belong to the set of Well Behaved Users. Many web sites, for example, care less about who their visitors are and more about whether they use the web site properly. In this case, you might authenticate their membership in the set by their ongoing interactions with your system.

One further issue you want to think about when considering using biometric authentication is whether there is any physical gap between where the biometric quantity is measured and where it is checked. In particular, checking biometric readings provided by an untrusted machine across the network is hazardous. What comes in across the network is simply a pattern of bits spread across one or more messages, whether it represents a piece of a web page, a phoneme in a VoIP conversation, or part of a scanned fingerprint. Bits is bits, and anyone can create any bit pattern they want. If an adversary knows what the bit pattern representing your fingerprint looks like, they may not need your finger, or even a fingerprint scanner, to create it and feed it to your machine. When the hardware performing the scanning is physically attached to your machine, there is less opportunity to slip in a spurious bit pattern that didn't come from the device. When the hardware is on the other side of the world on a machine you have no control over, there is a lot more opportunity. The point here is to be careful with biometric authentication information provided to you remotely.

In all, it sort of sounds like biometrics are pretty terrible for authentication, but that's the wrong lesson. For that matter, previous sections probably made it sound like all methods of authentication are terrible. Certainly none of them are perfect, but your task as a system designer is not to find the perfect authentication mechanism, but to use mechanisms that are well suited to the system and its environment. A good fingerprint reader built in to a smart phone might be pretty good at its job. A long, unguessable password can provide a decent amount of security. Well-designed smart cards can make it nearly impossible to authenticate yourself without having them in your hand. And where each type of mechanism fails, you can perhaps correct for that failure by using a second or third authentication mechanism that doesn't fail in the same cases.

Authenticating Non-Humans

No, we're not talking about aliens or extra-dimensional beings, or even your cat. If you think broadly about how computers are used today, you'll see that there are many circumstances in which no human user is associated with a process that's running. Consider a web server. There really isn't some human user logged in whose identity should be attached to the web server. Or think about embedded devices, such as a smart light bulb. Nobody logs in to a light bulb, but there is certainly code running there, and quite likely it is process-oriented code.

Mechanically, the operating system need not have a problem with the identities of such processes. Simply set up a user called `webserver` or `lightbulb` on the system in question and attach the identity of that "user" to the processes that are associated with running the web server or turning the light bulb on and off. But that does lead to the question of how you make sure that only real web server processes are tagged with that identity. We wouldn't want some arbitrary user on the web server machine creating processes that appear to belong to the server, rather than to that user.

One approach is to use passwords for these non-human users, as well. Simply assign a password to the web server user. When does it get used? When it's needed, which is when you want to create a process belonging to the web server, but you don't already have one in existence. The system administrator could log in as the web server user, creating a command shell and using it to generate the actual processes the server needs to do its business. As usual, the processes created by this shell process would inherit their parent's identity, `webserver`, in this case. More commonly, we skip the middleman (here, the login) and provide some mechanism whereby the privileged user is permitted to create processes that belong not to him, but to some other user, such as `webserver`. Alternately, we can provide a mechanism that allows a process to change its ownership, so the web server processes would start off under some other user's identity (such as the system administrator's) and change their ownership to `webserver`. Yet another approach is to allow a temporary change of process identity, while still remembering the original identity. (We'll say more about this last approach in a future chapter.) Obviously, any of these approaches require strong controls, since they allow one user to create processes belonging to another user.

As mentioned above, passwords are the most common authentication method used to determine if a process can be assigned to one of these non-human users. Sometimes no authentication of the non-human user is required at all, though. Instead, certain other users (like trusted system administrators) are given the right to assign new identities to the processes they create, without providing any further authentication information than their own. In Linux and other Unix systems, the `sudo` command offers this capability. For example,

```
sudo –u webserver apache2
```

would indicate that the `apache2` program should be started under the identity of `webserver`, rather than under the identity of whoever ran it. This command might require the user running it to provide his own authentication credentials (for extra certainty that it really is the privileged user asking for it, and not some random visitor accessing his computer during the privileged user's coffee break), but would not require authentication information associated with `webserver`. Any sub-processes created by `apache2` would, of course, inherit the identity of `webserver`.

One final identity issue we alluded to earlier is that sometimes we wish to identify not just individual users, but groups of users who share common characteristics, usually security-related characteristics. For example, we might have four or five system administrators, any one of whom is allowed to start up the web server. Instead of associating the privilege with each one individually, it's advantageous to create a system-meaningful group of users with that privilege. We would then indicate that the four or five administrators are members of that group. When one of them wished to do something requiring group membership, we would check that she was a member. We can either associate a group membership with each process, or use the process' individual identity information as an index into a list of groups that people belong to. The latter is more flexible, since it allows us to put each user into an arbitrary number of groups.

Most modern operating systems, including Linux and Windows, support these kinds of groups, since they provide ease and flexibility in dealing with application of security policies. They handle group membership and group privileges in manners largely analogous to those for individuals. For example, a child process will usually have the same group-related privileges as its parent. When working with such systems, it's important to remember that group membership provides a second path by which a user can obtain access to a resource, which has its benefits and its dangers.

Summary

If we want to apply security policies to actions taken by processes in our system, we need to know the identity of the processes, so we can make proper decisions. We start the entire chain of processes by creating a process at boot time belonging to some system user whose purpose is to authenticate users. They log in, providing authentication information in one or more forms to prove their identity. The system verifies their identity using this information and assigns their identity to a new process that allows the user to go about his business, which typically involves running other processes. Those other processes will inherit the user's identity from their parent process. Special secure mechanisms can allow identities of processes to be changed or to be set to something other than the parent's identity. The system can then be sure that processes belong to the proper user and can make security decisions accordingly.

Historically and practically, the authentication information provided to the system is either something the authenticating user knows (like a password or PIN), something he has (like a smart card or proof of possession of his smart phone), or something he is (like his fingerprint or voice scan). Each of these approaches has its strengths and weaknesses. A higher degree of security can be obtained by using multi-factor authentication, which requires a user to provide evidence in more than one form, such as requiring both a password and a one-time code that was texted to his smart phone.

References

[C00] Letter of recommendation to Tiberius Claudius Hermeros
Celer the Architect
Circa 100 A.D.
This letter introduced a slave to the imperial procurator, thus providing said curator evidence that the slave was who he claimed to be. You can read it in translation at http://papyri.info/ddbdp/c.ep.lat;;81.

[G13] "Anatomy of a hack: even your 'complicated' password is easy to crack"
Dan Goodin
http://www.wired.co.uk/article/password-cracking, May 2013.
A description of how three experts used dictionary attacks to guess a large number of real passwords, with 90% success.

[J] Judges 12, verses 5-6
The Bible
An early example of the use of biometrics. Failing this authentication had severe consequences, as the Gileadites slew mispronoucers, some 42,000 of them according to Judges.

[KA16] "VK.com Hacked! 100 Million Clear Text Passwords Leaked Online"
Swati Khandelwal
http://thehackernews.com/2016/06/vk-com-data-breach.html
One of many recent reports of stolen passwords stored in plaintext form.

[MT79] "Password Security: A Case History"
Robert Morris and Ken Thompson
Communications of the ACM, Vol. 22, No. 11, 1979.
A description of the use of passwords in early Unix systems. It also talks about password shortcomings from more than a decade earlier, in the CTSS system.

[M+02] "Impact of Artificial "Gummy" Fingers on Fingerprint Systems"
Tsutomu Matsumoto, Hiroyuki Matsumoto, Koji Yamada, and Satoshi Hoshino
SPIE Vol. #4677, January 2002.

A neat example of how simple ingenuity can reveal the security weaknesses of systems. In this case, the researchers showed how easy it was to fool commercial fingerprint reading machines.

[P46] "The Histories"
Polybius
Circa 146 B.C.
A history of the Roman Republic up to 146 B.C. Polybius provides a reasonable amount of detail not only about how the Roman Army used watchwords to authenticate themselves, but how they distributed them where they needed to be, which is still a critical element of using passwords.

[TR78] "On the Extraordinary: An Attempt at Clarification"
Marcello Truzzi
Zetetic Scholar, Vol. 1, No. 1, p. 11, 1978.
Truzzi was a scholar who investigated various pseudoscience and paranormal claims. He is unusual in this company in that he insisted that one must actually investigate such claims before dismissing them, not merely assume they are false because they conflict with scientific orthodoxy.

Access Control

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 part of this problem 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 solid, 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 something like

```
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 <u>is</u> 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?

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 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.

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) access 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[1], 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 hardware. If a process is given access to some virtual device, which is actually backed up by a real physical device controlled by the OS, 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

[1] 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.

appears to be theirs alone is actually shared, with the operating system running around behind the scenes to keep the illusion going. 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 abandon 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) 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 way would be to hire a massive intimidating doorman and give him a list of all the approved members. When someone wanted to enter the club, he would prove to the doorman who he was and the doorman would look him up on the list. If it was Linus Torvalds, the doorman would let him in, but he'd keep out the hoi polloi networking folks who had failed to distinguish themselves in the field of 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 architect with no OS chops wanted to get in, he wouldn't have the key and would be stuck outside. Compared to the other approach, we'd save on the salary of the doorman, though we would have to pay for the locks and keys[2]. 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 his 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

[2] 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 mask gets past our doorman, or if we aren't careful to determine that it really is Jerome Saltzer before handing a random guy 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.

approach based on the doorman 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.

Using ACLs for Access Control

What if, in the tradition of old British men's 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' 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 he isn't. If not, no access for him. If he is on the list, we'll typically go a step further to determine if the ACL entry for X allows the type of access he's requesting. 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 disk. Unless it's cached, we'll need to read it off the disk 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 disk 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 disk do we put the ACL to ensure that we at least don't also have to do another seek? It had better be close to 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 naïve 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 of his 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 naïve 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 9 bits for each file's ACL. 9 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 9 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 away in the inode. One is the members of a particular group or users; this group ID could also be stored in the inode. The final one is everybody else, 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 bit of 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 9 bit ACLs[3].

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

[3] 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 crossbreeding of these even earlier systems.

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.

ASIDE: NAME SPACES

We just encountered one of the interesting and difficult problems in distributed systems: what do names mean on different machines? The 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.

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 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 [DV64] being perhaps the earliest example. [W+74] describes the Hydra Operating System, which used capabilities as a fundamental control mechanism. [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 capability 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 <u>not</u> 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 off-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 our 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 in various places. 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 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 on whether its security policy permits B to access `/tmp/foo`, and to 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 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 some high overheads. If we used capabilities for file-based access control, a user might have tens or hundreds of thousands of capabilities, one for each file he 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 a user has at his discretion to give to each process he runs.

There is another option. Capabilities need not be stored in the operating system. Instead, they can be cryptographically protected entities, which solves the forgeability problem and allows them to be left in users' 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 his capability list. Revoking his access merely requires removing the capability from the list, which is easy enough if the operating system 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 inherits 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' 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 only 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 has ACL-based access control performed. 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' 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.

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 himself 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 access control elements, which allow further restriction (but not loosening) of the 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. [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.

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.

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 and services. Or, 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 organizations. 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 the company determines that all programmers should be granted access to a new library that has been developed, but accountants and janitors 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 (or, less happily, demoted to janitor), you can merely remove the *Programmer* role from the set he could take on.

RBAC sounds a bit like using groups in access control lists, and there is some similarity, but RBAC systems typically have a more formal approach than merely including individuals in groups. They 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. RBAC systems may offer finer granularity than merely being able to read or write a file. A particular role (*Salesman*, 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 restocking record for the same product to the same file, since salesmen don't do restocking. 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. There are implications for performance, storage of the security context information, and authentication that we hope are obvious to you, at this stage.

One can build a minimal RBAC system under Linux and similar OSes using ACLs and groups. The Linux `sudo` command offers a simple approach, allowing users with particular privileges to run commands under other identities. 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 the user who ran the command was on a system-maintained list of users allowed to take on the identity Programmer. Usually the `sudo` command requires a new authentication step, as with other RBAC systems.

For more advanced purposes, one typically uses a system that supports finer granularity and more careful tracking of role assignment. This 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, he could take on the role of Programmer to do the installation himself, but could not use that role to allow salesmen or accountants to perform the installation.

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 of them are malicious. Further, most apps have no legitimate need for most of the resources stored 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 thus implies that we should not give apps the full privileges belonging to the phone's owner. But they must have some privileges if they are to do anything interesting for that user

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 his device, he is shown what permissions it requires. He can either grant the app those permissions, not install the app, or limit its permissions, though the latter choice may also limit the app's utility. Also, the developer specifies ways in which other apps can communicate with his 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 the app's 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 that possession 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, though, again, limiting them may damage app functionality. Permission labels are a form of mandatory access control. The Android security model is discussed in detail in [E+09].

The Android security approach is interesting, but it is 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. Which is too bad if the app is malicious.

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"
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"
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"
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 here.

[FK92] "Role-Based Access Controls"
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*
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"
Peter Loscocco
Proceedings of the FREENIX Track:... 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 do so.

[S74] "Protection and Control of Information Sharing in Multics"
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"
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 . . .

[W+74] "Hydra: The Kernel of a Multiprocessor Operating System"
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.

Protecting Information With Cryptography

Introduction

In previous chapters, we've discussed clarifying your security goals, determining your security policies, using authentication mechanisms to identify principals, and using access control mechanisms to enforce policies concerning which principals can access which computer resources in which ways. While we identified a number of shortcomings and problems inherent in all of these elements of securing your system, if we regard those topics as covered, what's left for the operating system to worry about, from a security perspective? Why isn't that everything?

There are a number of reasons why we need more. Of particular importance, not everything is controlled by the operating system. But perhaps you respond, you told me the operating system is all-powerful! Not really. It has substantial control over a limited domain – the hardware on which it runs, using the interfaces it is given control of. It has no real control over what happens on other machines, nor what happens if one of its pieces of hardware is accessed via some mechanism outside the operating system's control.

But how can we expect the operating system to protect something when the system does not itself control access to that resource? The answer is to prepare the resource for trouble in advance. In essence, we assume that we are going to lose the data, or that an opponent will try to alter it improperly. And we take steps to ensure that such actions don't cause us problems. The key observation is that if an opponent cannot understand the data in the form he obtains it, our secrets are safe. Further, if he cannot understand it, he probably can't alter it, at least not in a controllable way. If he doesn't know what it means in its current form, how can he know how to change it into something he'd prefer?

The core technology we'll use is *cryptography*, a set of techniques to convert data from one form to another, in controlled ways with expected outcomes. We will convert the data from its ordinary form into another form using cryptography. If we do it right, the opponent will not be able to determine what the original data was by examining the protected form. Of course, if we ever want to use it again ourselves, we must be able to reverse that transformation and return the data to its ordinary form. That must be hard for the opponent to do, as well. If we can get to that point, we can also provide some protection for the data from alteration, or, more precisely, prevent the opponent from altering the data to suit his desires, and know when an opponent has tampered with our data. All through the joys of cryptography!

THE CRUX OF THE PROBLEM
HOW TO PROTECT INFORMATION OUTSIDE THE OPERATING SYSTEM'S DOMAIN?

How can we ensure that, even if others gain access to critical data outside the control of the operating system, they will neither be able to use nor alter it? What technologies are available to assist in this problem? How do we properly use those technologies? What are the limitations on what we can do with them?

Cryptography

Many books have been written about cryptography, but we're only going to spend a chapter on it. We'll still be able to say useful things about it because, fortunately, there are important and complex issues of cryptography that we can mostly ignore. That's because we aren't going to become cryptographers ourselves. We're merely going to be users of the technology, relying on experts in that esoteric field to provide us with tools that we can use without having full understanding of their workings[1]. That sounds kind of questionable, but you are already doing just that. Relatively few of us really understand the deep details of how our computer hardware works, yet we are able to make successful use of it, because we have good interfaces and know that smart people have taken great care in building the hardware for us. Similarly, cryptography provides us with strong interfaces, well-defined behaviors, and better than usual assurance that there is a lot of brain power behind the tools we use.

That said, cryptography is no magic wand, and there is a lot you need to understand merely to use it correctly. That, particularly in the context of operating system use, is what we're going to concentrate on here.

The basic idea behind cryptography is to take a piece of data and use an algorithm (often called a *cipher*), usually augmented with a second piece of information, to convert the data into a different form. The new form should look nothing like the old one, but, typically, we want to be able to run another algorithm, again augmented with a second piece of information, to convert the data back to its original form.

Let's formalize that just a little bit. We start with data P (which we usually call the *plaintext*), a second piece of information (which is called a *key*) K, and an encryption algorithm $E()$. We end up with C, the altered form of P, which we usually call the *ciphertext*:

$$C = E(P,K)$$

For example, we might take the plaintext "Transfer $100 to my savings account" and convert it into ciphertext "Sqzmredq #099 sn lx rzuhmfr zbbntms." This example

[1] If you'd like to learn more about the fascinating history of cryptography, check out [K96]. If more technical detail is your desire, [S96] is a good start.

actually uses a pretty poor encryption algorithm called a Caesar cipher. Spend a minute or two studying the plaintext and ciphertext and see if you can figure out what the encryption algorithm was in this case.

The reverse transformation takes *C*, which we just produced, a decryption algorithm *D()*, and the key *K*:

$$P=D(C,K)$$

So we can decrypt "Sqzmredq #099 sn lx rzuhmfr zbbntms" back into "Transfer $100 to my savings account." If you figured out how we encrypted the data in the first place, it should be easy to figure out how to decrypt it.

We use cryptography for a lot of things, but when discussing it generally, it's common to talk about messages being sent and received. In such discussions, the plaintext *P* is the message we want to send and the ciphertext *C* is the protected version of that message that we send out into the cold, cruel world.

For the encryption process to be useful, it must be deterministic, so the first transformation always converts a particular *P* using a particular *K* to a particular *C*, and the second transformation always converts a particular *C* using a particular *K* to the original *P*. In many cases, *E()* and *D()* are actually the same algorithm, but that is not required. Also, it should be very hard to figure out *P* from *C* without knowing *K*. Impossible would be nice, but we'll usually settle for computationally infeasible. If we have that property, we can show *C* to the most hostile, smartest opponent in the world and he still won't be able to learn what *P* is.

Provided, of course, . . .

This is where cleanly theoretical papers and messy reality start to collide. We only get that pleasant assurance of secrecy if the opponent does not know both *D()* and our key *K*. If he does, he'll apply *D()* and *K* to *C* and extract the same information *P* that we can.

It turns out that we usually can't keep *E()* and *D()* secret. Since we're not trying to be cryptographers, we won't get into the why of the matter, but it is extremely hard to design good ciphers. If the cipher has weaknesses, then an opponent can extract the plaintext *P* even without *K*. So we need to have a really good cipher, which is hard to come by. Most of us don't have a world-class cryptographer at our fingertips to design a new one, so we have to rely on one of a relatively small number of known strong ciphers. AES, a standard cipher that was carefully designed and thoroughly studied, is one good example that you should think about using.

It sounds like we've thrown away half our protection, since now the cryptography's benefit relies entirely on the secrecy of the key. Precisely. Let's say that again in all caps, since it's so important that you really need to remember it: THE CRYPTOGRAPHY'S BENEFIT RELIES ENTIRELY ON THE SECRECY OF THE KEY. It probably wouldn't hurt for you to re-read that statement a few dozen times, since the landscape is littered with insecure systems that did not take that lesson to heart.

The good news is that if you're using a strong cipher and are careful about maintaining key secrecy, your cryptography is strong. You don't need to worry about anything else. The bad news is that maintaining key secrecy in practical systems for real uses of cryptography isn't easy. We'll talk more about that later.

For the moment, revel in the protection we have achieved, and rejoice to learn that we've gotten more than secrecy from our proper use of cryptography! Consider the properties of the transformations we've performed. If our opponent gets access to our encrypted data, he can't understand it. But what if he can alter it? What he'll be altering is the encrypted form, so he'll make some changes in C to convert it to, say, C'. What will happen when we try to decrypt C? Well, it won't decrypt to P. It will decrypt to something else, say P'. For a good cipher of the type you should be using, it will be difficult to determine what a piece of ciphertext C' will decrypt to, unless you know K. That means it will be hard to predict which ciphertext you need to have to decrypt to a particular plaintext. Which in turn means that the attacker will have no idea what his altered ciphertext C' will decrypt to.

Out of all possible bit patterns it could decrypt to, the chances are good that P' will turn out to be garbage, when considered in the context of what we expected to see: ASCII text, a proper PDF file, or whatever. If we're careful, we can detect that P' isn't what we started with, which would tell us that our opponent tampered with our encrypted data. If we want to be really sure, we can perform a hashing function on the plaintext and include the hash in the message or encrypted file. If the plaintext we get out doesn't produce the same hash, we will have a strong indication that something is amiss.

ASIDE: DEVELOPING YOUR OWN CIPHERS

Don't.

It's tempting to leave it at that, since it's really important that you follow this guidance. But you may not believe it, so we'll expand a little. The world's best cryptographers often produce flawed ciphers. Are you one of the world's best cryptographers? If you aren't, and the top experts often fail to build strong ciphers, what makes you think you'll do better, or even as well?

We know what you'll say next: "but the cipher I wrote is so strong that I can't even break it myself." Well, pretty much anyone who puts their mind to it can create a cipher they can't break themselves. But remember those world-class cryptographers we talked about? How did they get to be world class? By careful study of the underpinnings of cryptography and by breaking other people's ciphers. They're very good at it, and if it's worth their trouble, they will break yours. Following which your secrets will be revealed, following which you will look foolish for designing your own cipher instead of using something standard like AES.

So, don't.

To be particularly careful, we can use a *cryptographic hash* for this purpose. Cryptographic hashes are designed to make it computationally infeasible to come up with two plaintexts that hash to the same value, so use of such a hash gives us extra assurance that our data hasn't been tampered with. In fact, if we only care about integrity, rather than secrecy, we can take the cryptographic hash of a piece of data, encrypt only the hash, and send both the encrypted hash and the data to our partner. If an opponent fiddles with the data in transit, when we decrypt the hash and repeat the hashing operation on the data, we'll see a mismatch and detect the tampering[2].

Cryptographic hashes can be used for other purposes, as well. Like other cryptographic algorithms, you're well advised to use standard algorithms for cryptographic hashing. For example, the SHA-3 algorithm is commonly regarded as a good choice.

So we can use cryptography to help us protect the integrity of our data, as well.

Wait, there's more! What if someone hands you a piece of data that has been encrypted with a key *K* that is known only to you and your buddy Remzi? You know you didn't create it, so if it decrypts properly using key *K*, you know that Remzi must have created it. After all, he's the only other person who knew key *K*, so only he could have performed the encryption. Voila, we have used cryptography for authentication! Unfortunately, cryptography will not clean your room, do your homework for you, or make thousands of julienne fries in seconds, but it's a mighty fine tool, anyway.

This form of cryptography is often called *symmetric cryptography*, because the same key is used to encrypt and decrypt the data. For a long time, everyone believed that was the only form of cryptography possible. It turns out everyone was wrong.

Public Key Cryptography

When we discussed using cryptography for authentication, you might have noticed a little problem. In order to verify the authenticity of a piece of encrypted information, you need to know the key used to encrypt it. If we only care about using cryptography for authentication, that's inconvenient. It means that we need to communicate the key we're using for that purpose to whoever might need to authenticate us. What if we're Microsoft, and we want to authenticate ourselves to every user who has purchased our software? We can't use just one key to do this, because we'd need to send that key to hundreds of millions of users and, once they had that key, they could pretend to be Microsoft by using it to encrypt information. Alternately, Microsoft could generate a different key for each of those hundreds of millions of users, but that would require

[2] Why do we need to encrypt the cryptographic hash? Well, anyone, including our opponent, can run a cryptographic hashing algorithm on anything, including his altered version of the message. If we don't encrypt the hash, he'll simply change the message, compute a new hash, replace both the original message and the original hash with his versions, and send the result. If the hash we sent is encrypted, though, he can't know what the encrypted version of the altered hash should be.

secretly delivering a unique key to hundreds of millions of users, not to mention keeping track of all those keys. Bummer.

Fortunately, our good friends, the cryptographic wizards, came up with a solution. What if we use two different keys for cryptography, one to encrypt and one to decrypt? Our encryption operation becomes

$$C = E(P,K_{encrypt})$$

And our decryption operation becomes

$$P=D(C,K_{decrypt})$$

Life has just become a lot easier for Microsoft. They can tell everyone their decryption key $K_{decrypt}$, but keep their encryption key $K_{encrypt}$ secret. They can now authenticate their data by encrypting it with their secret key, while their hundreds of millions of users can check the authenticity using the key Microsoft made public. For example, Microsoft could encrypt an update to their operating system with $K_{encrypt}$ and send it out to all their users. Each user could decrypt it with $K_{decrypt}$. If it decrypted into a properly formatted software update, the user could be sure it was created by Microsoft. Since no one else knows that private key, no one else could have created the update.

Sounds like magic, but it isn't. It's actually mathematics coming to our rescue, as it so frequently does. We won't get into the details here, but you have to admit it's pretty neat. This form of cryptography is called *public key cryptography*, since one of the two keys can be widely known to the entire public, while still achieving desirable results. The key everyone knows is called the *public key*, and the key that only the owner knows is called the *private key*. Public key cryptography (often abbreviated as PK) has a complicated invention history, which, while interesting, is not really germane to our discussion. Check out a paper by a pioneer in the field, Whitfield Diffie, for details [D88].

Public key cryptography avoids one hard issue that faced earlier forms of cryptography: securely distributing a secret key. Here, the private key is created by one party and kept secret by him. It's never distributed to anyone else. The public key must be distributed, but generally we don't care if some third party learns this key, since they can't use it to sign messages. Distributing a public key is an easier problem than distributing a secret key, though, alas, it's harder than it sounds. We'll get to that.

Public key cryptography is actually even neater, since it works the other way around. You can use the decryption key $K_{decrypt}$ to encrypt, in which case you need the encryption key $K_{encrypt}$ to decrypt. We still expect the encryption key to be kept secret and the decryption key to be publically known, so doing things in this order no longer allows authentication. Anyone could encrypt with $K_{decrypt}$, after all. But only the owner of the key can decrypt such messages using $K_{encrypt}$. So that allows anyone to send an encrypted message to someone who has a private key, provided you know their public key. Thus, PK allows authentication if you encrypt with the private key and secret communication if you encrypt with the public key.

What if you want both, as you very well might? You'll need two different key pairs to do that. Let's say Alice wants to use PK to communicate secretly with her pal Bob, and also wants to be sure Bob can authenticate her messages. Let's also say Alice and Bob each have their own PK pair. Each of them knows his or her own private key and the other party's public key. If Alice encrypts her message with her own private key, she'll authenticate the message, since Bob can use her public key to decrypt and will know that only Alice could have created that message. But everyone knows Alice's public key, so there would be no secrecy achieved. However, if Alice takes the authenticated message and encrypts it a second time, this time with Bob's public key, she will achieve secrecy as well. Only Bob knows the matching private key, so only Bob can read the message. Of course, Bob will need to decrypt twice, once with his private key and then a second time with Alice's public key.

Sounds expensive. It's actually worse than you think, since it turns out that public key cryptography has a shortcoming: it's much more computationally expensive than traditional cryptography that relies on a single shared key. Public key cryptography can take hundreds of times longer to perform than standard symmetric cryptography. As a result, we really can't afford to use public key cryptography for everything. We need to pick and choose our spots, using it to achieve the particular things it's so good at.

There's another important issue. We rather blithely said that Alice knows Bob's public key and Bob knows Alice's. How did we achieve this blissful state of affairs? Originally, only Alice knew her public key and only Bob knew his public key. We're going to need to do something to get that knowledge out to the rest of the world if we want to benefit from the magic of public key cryptography. And we'd better be careful about it, since Bob is going to assume that messages encrypted with the public key he thinks belongs to Alice were actually created by Alice. What if some evil genius, called, perhaps, Eve, manages to convince Bob that Eve's public key actually belongs to Alice? If that happens, then messages created by Eve would be misidentified by Bob as originating from Alice, totally subverting our entire goal of authenticating the messages. So we'd better make sure Eve can't fool Bob about which public key belongs to Alice.

This leads down a long and rather shadowy road to the arcane realm of key distribution infrastructures. You will be happier if you don't try to travel that road yourself, since even the most well prepared pioneers who have hazarded it often come to grief. We'll talk a bit more about how, in practice, we distribute public keys in our chapter on distributed system security. For the moment, bear in mind that the beautiful magic of public key cryptography rests on the grubby and uncertain foundation of key distribution.

One more thing about PK cryptography: THE CRYPTOGRAPHY'S BENEFIT RELIES ENTIRELY ON THE SECRECY OF THE KEY. (Bet you've heard that before.) In this case, the private key. But the secrecy of that private key is every bit as important to the overall benefit of public key cryptography as the secrecy of the single shared key in the case of symmetric cryptography. Never divulge private keys. Never share private keys. Take great care in your use of private keys and in how you store them. If you lose a private key, everything you used it for is at risk, and whoever gets hold of it can pose as you and read your secret messages. That wouldn't be very good, would it?

Cracking Cryptography

Chances are that you've heard about people cracking cryptography. It's a popular theme in film and television. How worried should you be about that?

Well, if you didn't take our earlier advice and went ahead and built your own cipher, you should be very worried. Worried enough that you should stop reading this, rip out your own cipher from your system, and replace it with a well-known respected standard. Go ahead, we'll still be here when you get back.

What if you did use one of those standards? In that case, you're probably OK. If you use a modern standard, with a few unimportant exceptions, there are no known ways to read data encrypted with these algorithms without obtaining the key. Which isn't to say your system is secure, but probably no one will break into it by cracking the cryptographic algorithm.

How will they do it, then? Probably by exploiting software flaws in your system having nothing to do with the cryptography, but there's some chance they will crack it by obtaining your keys or exploiting some other flaw in your management of cryptography. How? Software flaws in how you create and use your keys are a common problem. In distributed environments, flaws in the methods used to share keys are also a common weakness that can be exploited. Peter Gutmann produced a nice survey of the sorts of problems improper management of cryptography frequently causes [G02]. More recently, the Heartbleed attack demonstrated a way to obtain keys being used in OpenSSL sessions from the memory of a remote computer, which allowed an attacker to decrypt the entire session, despite no flaws in either the cipher itself or its implementation. This flaw allowed attackers to read the traffic of something between ¼ and ½ of all sites using HTTPS, the cryptographically protected version of HTTP [D+14].

One way attackers deal with cryptography is by guessing the key. Doing so doesn't actually crack the cryptography at all. Cryptographic algorithms are designed to prevent people who don't know the key from obtaining the secrets. If you know the key, it's not supposed to make decryption hard.

So an attacker could try simply guessing each possible key and trying it. That's called a *brute force attack*, and it's why you should use long keys. For example, AES keys are at least 128 bits. Assuming you generate your AES key at random, an attacker will need to make 2^{127} guesses at your key, on average, before he gets it right. That's a lot of guesses and will take a lot of time. Of course, if a software flaw causes your system to select one out of thirty two possible AES keys, instead of one out of 2^{128}, a brute force attack may become trivial. Key selection is a big deal for cryptography.

For example, the original 802.11 wireless networking standard included no cryptographic protection of data being streamed through the air. The first attempt to add such protection was called WEP (Wired Equivalent Protocol, a rather optimistic name). WEP was constrained by the need to fit into the existing standard, but the method it used to generate and distribute symmetric keys was seriously flawed. Merely by listening in on wireless traffic on an 802.11 network, an attacker could determine the key being used in as little as a minute. There are widely available tools that allow anyone to do so.

As another example, an early implementation of the Netscape web browser generated cryptographic keys using some easily guessable values as seeds to a random number generator, such as the time of day and the ID of the process requesting the key. Researchers discovered they could guess the keys produced in around 30 seconds [GW96].

You might have heard that PK systems use much longer keys, 2K or 4K bits. Sounds much safer, no? Shouldn't that at least make them stronger against brute force attacks? However, you can't select keys for this type of cryptosystem at random. Only a relatively few pairs of public and private keys are possible. That's because the public and private keys must be related to each other for the system to work. The relationship is usually mathematical, and usually intended to be mathematically hard to derive, so knowing the public key should not make it easy to know the private key. However, with the public key in hand, one can use the mathematical properties of the system to derive the private key eventually. That's why PK systems use such big keys – to make sure "eventually" is a very long time.

TIP: SELECTING KEYS

One important aspect of key secrecy is selecting a good one to begin with. For public key cryptography, you need to run an algorithm to select one of the few possible pairs of keys you will use. But for symmetric cryptography, you are free to select any of the possible keys. How should you choose?

Randomly. If you use any deterministic method to select your key, your opponent's problem of finding out your key has just been converted into a problem of figuring out your method. Worse, since you'll probably generate many keys over the course of time, once he knows your method, he'll get all of them. If you use random chance to generate keys, though, figuring out one of them won't help your opponent figure out any of your other keys.

Unfortunately, true randomness is hard to come by. The best source for operating system purposes is to examine hardware processes that are believed to be random in nature and convert the results into random numbers. That's called *gathering entropy*. In Linux, this is done for you automatically, and you can use the gathered entropy by reading `/dev/random`. Use it to generate your keys.

But that only matters if you keep the private key secret. By now, we hope this sounds obvious, but many makers of embedded devices use PK to provide encryption for those devices, and include a private key in the device's software. All too often, the same private key is used for all devices of a particular model. Such shared private keys invariably become, well, public. In September 2016, one study found 4.5 million embedded devices relying on these private keys that were no longer so private [V16]. Anyone could pose as any of these devices for any purpose, and could read any information sent to them using PK. In essence, the cryptography performed by these devices was little more than window dressing and did not increase the security of the devices by any appreciable amount.

To summarize, cracking cryptography is usually about learning the key. So . . .

THE CRYPTOGRAPHY'S BENEFIT RELIES ENTIRELY ON THE SECRECY OF THE KEY.

Cryptography and Operating Systems

Cryptography is fascinating, but lots of things are fascinating, while having no bearing on operating systems. Why did we bother spending half a chapter on cryptography? Because we can use it to protect operating systems.

But not just anywhere and for all purposes. We've pounded into your head that key secrecy is vital for effective use of cryptography. That should make it clear that any time the key can't be kept secret, you can't effectively use cryptography. Casting your mind back to the first chapter on security, remember that the operating system has control of and access to all resources on a computer. Which implies that if you have encrypted information on the computer, and you have the necessary key to decrypt it on the same computer, the operating system on that machine can decrypt the data, whether that was the effect you wanted or not.

Either you trust your operating system or you don't. If you don't, life is going to be unpleasant anyway, but one implication is that the untrusted operating system, having access at one time to your secret key, can copy it and re-use it whenever it wants to. If, on the other hand, you trust your operating system, you don't need to hide your data from it, so cryptography isn't necessary in this case. This observation has relevance to any situation in which you provide your data to something you don't trust. For instance, if you don't trust your cloud computing facility with your data, you won't improve the situation by giving the data to them in plaintext and asking them to encrypt it. They've seen the plaintext and can keep a copy of the key[3].

If you're sure your operating system is trustworthy right now, but are concerned it might not be later, you can encrypt something now and make sure the key is not stored on the machine. Of course, if you're wrong about the current security of the operating system, or if you ever decrypt the data on the machine after the OS goes rogue, your

[3] There's one possible exception worth mentioning. Those cryptographic wizards have created a form of cryptography called homomorphic cryptography, which allows you to perform operations on the encrypted form of the data without decrypting it. For example, you could add one to an encrypted integer without decrypting it first. When you decrypted the result, sure enough, one would have been added to the original number. Homomorphic ciphers have been developed, but high computational and storage costs render them impractical for most purposes, as of the writing of this chapter. Perhaps that will change, with time.

cryptography will not protect you, since that ever-so-vital secrecy of the key will be compromised.

So if cryptography won't protect us against a dishonest operating system, what's operating system uses for cryptography are there? We saw a specialized example in the chapter on authentication. Some cryptographic operations are one-way: they can encrypt, but never decrypt. We can use these to securely store passwords in encrypted form, even if the operating system is compromised, since the encrypted passwords can't be decrypted. (But if the legitimate user ever provides the correct password to a compromised operating system, all bets are off, alas. The compromised operating system will copy the password provided by the user and hand it off to whatever villain is working behind the scenes, before it runs the password through the one-way cryptographic hashing algorithm.)

What else? In a distributed environment, if we encrypt data on one machine and then send it across the network, all the intermediate components won't be part of our machine, and thus won't have access to the key. The data will be protected in transit. Of course, our partner on the final destination machine will need the key if she is to use the data. As we promised before, we'll get to that issue in another chapter.

Anything else? Well, what if someone can get access to some of our hardware without going through our operating system? If the data stored on that hardware is encrypted, and the key isn't on that hardware itself, the cryptography will protect the data. This form of encryption is sometimes called at-rest data encryption, to distinguish it from encrypting data we're sending between machines. It's useful and important, so let's examine it in more detail.

At-Rest Data Encryption

As we saw in the chapters on persistence, data can be stored on a disk drive, flash drive, or other medium. If it's sensitive data, we might want some of our desirable security properties, such as secrecy or integrity, to be applied to it. One technique to achieve these goals for this data is to store it in encrypted form, rather than in plaintext. Of course, encrypted data cannot be used in most computations, so if the machine where it is stored needs to perform a general computation on the data, it must first be decrypted. If the purpose is merely to preserve a safe copy of the data, rather than to use it, decryption may not be necessary, but that is not the common case.

The data can be encrypted in different ways, using different ciphers (DES, AES, Blowfish), at different granularities (records, data blocks, individual files, entire file systems), by different system components (applications, libraries, file systems, device drivers). One common general use of at-rest data encryption is called "full disk encryption." This usually means that the entire contents (or almost the entire contents) of the storage device are encrypted. Despite the name, full-disk encryption can actually be used on many kinds of persistent storage media, not just hard disk drives. Full disk encryption is usually provided either in hardware (built into the storage device) or by system software (a device driver or some element of a file system). In either case, the operating system plays a role in the protection provided. Windows BitLocker and Apple's FileVault are examples of software-based full disk encryption.

Generally, at boot time either the decryption key or information usable to obtain that key (such as a passphrase – like a password, but possibly multiple words) is requested from the user. If the right information is provided, the key or keys necessary to perform the decryption become available (either to the hardware or the operating system). As data is placed on the device, it is encrypted. As data moves off the device, it is decrypted. The data remains decrypted as long as it is stored anywhere in the machine's memory, including in shared buffers or user address space. When new data is to be sent to the device, it is first encrypted. The data is never placed on the storage device in decrypted form. After the initial request to obtain the decryption key is performed, encryption and decryption are totally transparent to users and applications. They never see the data in encrypted form and are not asked for the key again, until the machine reboots.

Cryptography is a computationally expensive operation, particularly if performed in software. There will be overhead associated with performing software-based full disk encryption. Reports of the amount of overhead vary, but a few percent extra latency for disk-heavy operations is common. For operations making less use of the disk, the overhead may be imperceptible. For hardware based full disk encryption, the rated speed of the disk drive will be achieved, which may or may not be slower than a similar model not using full disk encryption.

What does this form of encryption protect against?

- It offers no extra protection against users trying to access data they should not be allowed to see. Either the standard access control mechanisms that the operating system provides work (and such users can't get to the data because they lack access permissions) or they don't (in which case such users will be given equal use of the decryption key as anyone else).
- It does not protect against flaws in applications that divulge data. Such flaws will permit attackers to pose as the user, so if the user can access the unencrypted data, so can the attacker. So, for example, it offers little protection in the face of buffer overflow or SQL injection attacks.
- It does not protect against dishonest privileged users on the system, such as a system administrator. If his privileges allow him to pose as the user who owns the data or to install system components that give him access to the user's data, he will be given decrypted copies of the data on request.
- It does not protect against security flaws in the operating system itself. Once the key is provided, it is available (directly in memory, or indirectly by asking the hardware to use it) to the operating system, whether that OS is trustworthy and secure or compromised and insecure.

So what benefit does this form of encryption provide? Consider this situation. If a hardware device storing data is physically moved from one machine to another, the operating system on the other machine is not obligated to honor the access control information stored on the device. In fact, it need not even use the same file system to access that device. For example, it can treat the device as merely a source of raw data blocks, rather than an organized file system. So any access control information associated with files on the device might be ignored by the new operating system.

However, if the data on the device is encrypted via full disk encryption, the new machine will usually be unable to obtain the encryption key. It can access the raw blocks, but they are encrypted and cannot be decrypted without the key. This benefit would be useful if the hardware in question was stolen and moved to another machine, for example. This situation is a very real possibility for mobile devices, which are frequently lost or stolen. Disk drives are sometimes resold, and data belonging to the former owner (including quite sensitive data) has been found on them by the re-purchaser. These are important cases where full disk encryption provides real benefits.

For other forms of encryption of data at rest, the system must still address the issues of how much is encrypted, how to obtain the key, and when to encrypt and decrypt the data, with different types of protection resulting depending on how these questions are addressed. Generally, such situations require that some software ensures that the unencrypted form of the data is no longer stored anywhere, including caches, and that the cryptographic key is not available to those who might try to illicitly access the data. There are relatively few circumstances where such protection is of value, but there are a few common examples:

- Archiving data that might need to be copied and must be preserved, but need not be used. In this case, the data can be encrypted at the time of its creation, and perhaps never decrypted, or only decrypted under special circumstances under the control of the data's owner. If the machine was uncompromised when the data was first encrypted and the key is not permanently stored on the system, the encrypted data is fairly safe.
- Storing sensitive data in a cloud computing facility, a variant of the previous example. If one does not completely trust the cloud computing provider (or one is uncertain of how careful that provider is), encrypting the data before sending it to the cloud facility is wise. Many cloud backup products include this capability. In this case, the cryptography and key use occur before moving the data to the untrusted system, or after it is recovered from that system.
- User-level encryption performed through an application. For example, a user might choose to encrypt an email message, with any stored version of it being in encrypted form. In this case, the cryptography will be performed by the application, and the user will do something to make a cryptographic key available to the application. Ideally, that application will ensure that the unencrypted form of the data and the key used to encrypt it are no longer readily available after encryption is completed. Remember, however, that while the key exists, the operating system can obtain access to it without your application knowing.

One important special case for encrypting selected data at rest is a password vault (also known as a key ring). Typical users interact with many remote sites that require them to provide passwords. (Authentication based on what you know, remember?) The best security is achieved if one uses a different password for each site, but doing so places a burden on the human user, who generally has a hard time remembering many passwords. A solution is to encrypt all the different passwords and store them on the machine, indexed by the site they are used for. When one of the passwords is required, it is decrypted and provided to the site that requires it.

For password vaults and all such special cases, the system must have some way of obtaining the key whenever data needs to be encrypted or decrypted. If an attacker can obtain the key, the cryptography becomes useless, so safe storage of the key becomes critical. Typically, if the key is stored in unencrypted form anywhere on the computer in question, the encrypted data is at risk, so well designed encryption systems tend not to do so. For example, in the case of password vaults, the key used to decrypt the passwords is not stored in the machine's stable storage. It is obtained by asking the user for it when required, or asking him for a passphrase used to derive the key. The key is then used to decrypt the needed password. Maximum security would suggest destroying the key as soon as this decryption was performed (remember the principle of least privilege?), but doing so would imply that the user would have to re-enter the key each time he needed a password. A compromise between usability and security is reached, in most cases, by remembering the key after first entry for a significant period of time, but only keeping it in RAM. When the user logs out, or the system shuts down, or the application that handles the password vault (such as a web browser) exits, the key is "forgotten." This approach is reminiscent of single sign-on systems, where a user is asked for his password when he first accesses the system, but is not required to re-authenticate himself again until he logs out. It has the same disadvantages as those systems, such as permitting an unattended terminal to be used by unauthorized parties to use someone else's access permissions.

Summary

Cryptography can offer certain forms of protection for data even when that data is no longer in a system's custody. These forms of protection include secrecy, integrity, and authentication. Cryptography achieves such protection by converting the data's original bit pattern into a different bit pattern, using an algorithm called a cipher. In most cases, the transformation can be reversed to obtain the original bit pattern. Symmetric ciphers uses a single secret key shared by all parties with rights to access the data. Asymmetric ciphers use one key to encrypt the data and a second key to decrypt the data, with one of the keys kept secret and the other commonly made public. Strong ciphers make it computationally infeasible to obtain the original bit pattern without access to the required key.

For operating systems, the obvious situations in which cryptography can be helpful are when data is sent to another machine, or when hardware used to store the data might be accessed without the intervention of the operating system. In the latter case, data can be encrypted on the device (using either hardware or software), and decrypted as it is delivered to the operating system.

Ciphers are generally not secret, but rather are widely known and studied standards. A cipher's ability to protect data thus relies entirely on key secrecy. If attackers can learn, deduce, or guess the key, all protection is lost. Thus, extreme care in key selection and maintaining key secrecy is required if one relies on cryptography for protection.

References

[D88] "The First Ten Years of Public Key Cryptography"
Whitfield Diffie
Communications of the ACM, Vol. 76, No. 5, May 1988.
A description of the complex history of where public key cryptography came from.

[D+14] "The Matter of Heartbleed"
Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson
Proceedings of the 2014 Conference on Internet Measurement Conference.
A good description of the Heartbleed vulnerability in OpenSSL and its impact on the Internet as a whole. Worth reading for the latter, especially, as it points out how one small bug in one critical piece of system software can have a tremendous impact.

[G02] "Lessons Learned in Implementing and Deploying Crypto Software"
Peter Gutmann
Usenix Security Symposium, 2002
A good analysis of the many ways in which poor use of a perfectly good cipher can totally compromise your software, backed up by actual cases of the problems occurring in the real world.

[GW96] "Randomness and the Netscape Browser"
Ian Goldberg and David Wagner
Dr. Dobbs Journal, January 1996.
Another example of being able to deduce keys that were not properly created and handled, in this case by guessing the inputs to the random number generator used to create the keys. Aren't attackers clever?

[K96] *The Codebreakers*
David Kahn
Scribner Publishing, 1996.
A long history of cryptography, its uses, and how it is attacked.

[S96] *Applied Cryptography*
Bruce Schneier
Jon Wiley and Sons, Inc. 1996.
A detailed description of how to use cryptography in many different circumstances, including example source code.

[V16] "House of Keys: 9 Months later... 40% Worse"
Stefan Viehböck
http://blog.sec-consult.com/2016/09/house-of-keys-9-months-later-40-worse.html

A web page describing the unfortunate ubiquity of the same private key being used in many different embedded devices.