52

A Dialogue on Security

Chapter by **Peter Reiher (UCLA)**

Professor: *Hello again, student!*

Student: *I thought we were done with all this. We've already had three pillars, and I even stuck around for a few appendices. Will I never be done with this class?*

Professor: *That depends on who I am. Some professors want to talk about security and some don't. Unfortunately for you, given that you're here, I'm one of those who want to.*

Student: *OK, I suppose we'd better just get on with it.*

Professor: *That's the spirit! Soonest begun, soonest done. So, let's say you have a peach...*

Student: *You told me we were at least done with peaches!*

Professor: *When one is discussing security, lies will always be a part of the discussion. Anyway, you've got a peach. You certainly wouldn't want to turn around and find someone had stolen your peach, would you?*

Student: *Well, if it isn't as rotten as the one you ended up with, I suppose not.*

Professor: *And you probably wouldn't be any happier if you turned around and discovered someone had swapped out your peach for a turnip, either, would you?*

Student: *I guess not, though I do know a couple of good recipes for turnips.*

Professor: *And you also wouldn't want somebody slapping your hand away every time you reached for your peach, right?*

Student: *No, that would be pretty rude.*

Professor: *You wouldn't want that happening to any of the resources your computer controls, either. You might be even unhappier, if they're really important resources. You wouldn't want the love letter you're in the middle of composing to leak out, you wouldn't want someone to reset the saved state in your favorite game to take you back to the very beginning, and you would be mighty upset if, at midnight the evening before your project was due, you weren't allowed to log into your computer.*

Student: *True, those would all pretty much suck.*

Professor: *Let's try to keep a professional tone here. After all, this is a classroom. Kind of. That's what operating system security is all about, and that's what I'm here to tell you about. How can you ensure that secrets remain confidential? How can you guarantee the integrity of your important data? How can you ensure that you can use your computer resources when you want to? And these questions apply to all of the resources in your computer, all the time, forever.*

Student: *All this sounds a little like reliability stuff we talked about before...*

Professor: *Yes and no. Bad things can happen more or less by accident or through poor planning, and reliability is about those sorts of things. But we're going a step further. SOMEBODY WANTS YOUR PEACH!!!!*

Student: *Stop shouting! You were the one asking for a professional tone.*

Professor: *My apologies, I get excited about this stuff sometimes. The point I was trying to make is that when we talk about security, we're talking about genuine adversaries, human adversaries who are trying to make things go wrong for you. That has some big implications. They're likely to be clever, malevolent, persistent, flexible, and sneaky. You may already feel like the universe has it in for you (most students feel that way, at any rate), but these folks really, truly are out to get you. You're going to have to protect your assets despite anything they try.*

Student: *This sounds challenging.*

Professor: *You have no idea... But you will! YOU WILL!! (maniacal laughter)*

53

Introduction to Operating System Security

Chapter by **Peter Reiher (UCLA)**

53.1 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, and there are relatively few choices of which to run. 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.

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 adversary 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[1]. 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 adversaries, 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. 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 every process could be trusted to do anything it wants with any hardware resource and any piece of data on the machine without harming any other process, 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 unwriteable does not get altered. If the applications cannot rely on proper implementation of security guarantees for OS abstractions, then they cannot use these abstractions to achieve their own security goals. At the minimum, that implies a great deal more work on

[1] If you don't understand it, you have a lot of re-reading to do. A lot.

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 or a well-defined scheduling policy) cannot be trusted.

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 you might not think you're interested in security, but you need to understand what your OS does to secure itself to also understand how to get the system to do what you want.

CRUX: HOW TO 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?

53.2 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 (or almost 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's 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

ASIDE: SECURITY ENCLAVES

A little bit back, we said the operating system controls "almost all" the hardware on the machine. That kind of caveat should have gotten you asking, "well, what parts of the hardware doesn't it control?" Originally, it really was all the hardware. But starting in the 1990s, hardware developer began to see a need to keep some hardware isolated, to a degree, from the operating system. The first such hardware was primarily intended to protect the boot process of the operating system. **TPM**, or **Trusted Platform Module**, provided assurance that you were booting the version of the operating system you intended to, protecting you from attacks that tried to boot compromised versions of the system. More recently, more general hardware elements have tried to control what can be done on the machine, typically with some particularly important data, often data that is related to cryptography. Such hardware elements are called security enclaves, since they are meant to allow only safe use of this data, even by the most powerful, trusted code in the system – the operating system itself. They are often used to support operations in a cloud computing environment, where multiple operating systems might be running under virtual machines sharing the same physical hardware.

This turns out to be a harder trick than anyone expected. Security tricks usually are. Security enclaves often prove not to provide quite as much isolation as their designers hoped. But the attacks on them tend to be sophisticated and difficult, and usually require the ability to run privileged code on the system already. So even if they don't achieve their full goals, they do put an extra protective barrier against compromised operating system code.

- give any process access to any other process's 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[2], 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 completely compromise everything about the machine the system runs on, so preventing them and patching any that are found is vitally important.

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

53.3 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** – 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 adversary 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. One important aspect of integrity is authenticity. It's often important to be sure not only that information has not changed, but that it was created by a particular party and not by an adversary.

- **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.

An important extra dimension of all three of these goals is that we want controlled sharing in our systems. We share our secrets with some people and not with others. We allow some people to change our enterprise's databases, but not just anyone. Some systems need to be made available to a particular set of preferred users (such as those who have paid to play your on-line game) and not to others (who have not). Who's doing the asking matters a lot, in computers as in everyday life.

Another important aspect of security for computer systems is we often want to be sure that when someone told us something, they cannot later deny that they did so. This aspect is often called **non-repudiation**. The

harder and more expensive it is for someone to repudiate their actions, the easier it is to hold them to account for those actions, and thus the less likely people are to perform malicious actions. After all, they might well get caught and will have trouble denying they did it.

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's 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.

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 highly 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.

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 system precisely what that policy is. With some important exceptions (like maintaining a process's address space private unless specifically directed otherwise), the operating system merely supplies general mechanisms that can implement many specific policies. Without intelligent design of poli-

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's 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 really want to 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 the possibilities of legitimate sharing they open are also potential channels for malicious attacks. Finally, the OS does not always have complete control of the hardware...

cies and careful application of the mechanisms, however, what the operating system *should* or *could* do may not be what your operating system *will* do.

53.4 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 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 its 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 adversary 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.

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.

[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.

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 Seacord [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 details of the language. For a comprehensive treatment on how to evaluate if your system is secure, start with Dowd et al.'s work [D+07].

53.5 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 physical 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. (The software folks among us should remember to regularly thank the hardware folks for all the great stuff they've given us to work with.)

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 chap-

TIP: BE CAREFUL OF 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.

Generally, thinking about security is a bit different than thinking about many other system design issues. It's more adversarial. If you want to learn more about good ways to think about security of the systems you build, check out Schneier's book "Secrets and Lies" [SC00].

ters. 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.

53.6 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 for all systems.

The next question to address is, what mechanisms should our operating system provide 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" by 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.*

[SC00] "Secrets and Lies" by Bruce Schneier. Wiley Computer Publishing, 2000. *A good high-level perspective of the challenges of computer security, developed at book length. Intended for an audience of moderately technically sophisticated readers, and well regarded in the security community. A must-read if you intend to work in that field.*

[SE13] "Secure Coding in C and C++" by 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" by 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.*

54

Authentication

Chapter by **Peter Reiher (UCLA)**

54.1 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 your 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. In computer security discussions, we often refer to the party asking for something as the **principal**. Principals are security-meaningful entities that can request access to resources, such as human users, groups of users, or complex software systems.

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 made by particular processes, which trap from user code into the operating system. The operating system then takes control and performs some service in response to the system call. Associated with the 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 computer security discussions, the process or other active computing entity performing the request on behalf of a principal is often called its **agent**.

The request is for access to some particular resource, which we frequently refer to as the **object** of the access request[1]. Either the operating system has already determined this agent process can access the object or it hasn't. If it has determined that the process is permitted access, the OS can remember that decision and it's merely a matter of keeping track, presumably in some per-process data structure like the PCB, of that fact. For example, as we discovered when investigating virtualization of memory, per-process data structures like page tables show which pages and page frames can be accessed by a process at any given time. Any form of data created and managed by the operating system that keeps track of such access decisions for future reference is often called a **credential**.

If the operating system has not already produced a credential showing that an agent process can access a particular object, however, it needs information about the identity of the process's principal to determine if its request should be granted. Different operating systems have used different types of identity for principals. 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 its power, 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[2]. 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

[1] Another computer science overloading of the word "object." Here, it does not refer to "object oriented," but to the more general concept of a specific resource with boundaries and behaviors, such as a file or an IPC channel.

[2] 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.

misidentify a programmer employee process as an accounting department employee process, you could end up with an empty bank account. (Not to mention needing to hire a new programmer.) Or if you fail to identify your company president correctly when he or she is trying to give an important presentation to investors, you may find yourself out of a job once the company determines that you're the one who derailed the next round of startup capital, because the system didn't allow the president to access the presentation that would have bowled over some potential investors.

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. But we need to bear in mind one other important characteristic of operating systems' usual approach to authentication: once a principal has been authenticated, systems will almost always rely on that authentication decision for at least the lifetime of the process. This characteristic puts a high premium on getting it right. Mistakes won't be readily corrected. Which leads to the crux:

CRUX: HOW TO SECURELY IDENTIFY PROCESSES

For systems that support processes belonging to multiple principals, 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 improperly change the identity of a process?

54.2 Attaching Identities To Processes

Where do processes come from? Usually they are created by other processes. One simple way to attach an identity to a new process, then, is to copy the identity of the 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 a 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, we're not going to be able to implement any policy that differentiates the privileges of one process versus another.

We must arrange that some processes have different identities and use those differences to manage our security policies. Consider a multi-user system. We can assign identities to processes 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.

If processes have a security-relevant identity, like a user ID, we're going to have to set the proper user ID for a new process. In most systems, a user has a process that he or she 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 that user. Since the operating system can fiddle with its own data structures, like the process control block, it can set the new process's ownership to the user who just joined the system.

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 the user 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 building a towering security structure with unforeseeable implications based on the OS making the right decision here, so it's important. What are our options?

54.3 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 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 this attempt to sneak in. Second, if he or she is an authorized user, we need to determine, which one?

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

- Authentication based on what you know
- Authentication based on what you have
- 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 [P-46], 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 given to an imperial procurator at some time in the 2nd century AD [C100] – authentication based on what the slave had. 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 [JB-500]. This was a form of authentication by what you are: a native speaker of the Gileadites' dialect or 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.

54.4 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 their 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 must 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 might 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 this person based on the person 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[3].

[3] "Might" is too weak a word. The first known incident of such stored passwords leaking is from 1962 [MT79]; such leaks happen to this day with depressing regularity and much larger scope. [KA16] discusses a leak of over 100 million passwords stored in usable 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, logs, 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. In some cases, even secrets only kept in the heap of an executing program have been divulged, so avoid storing and keeping secrets even in running programs.

Interestingly enough, though, our system does not actually 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 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 or she claims to be the password, hash the claim and compare it to the stored hashed value. If it matches, you believe he or she 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. If the attacker provides the hash, instead of the password, the hash itself gets hashed by the system, and a hash of a hash won't match the hash.

There is a little more to it than that. The benefit we're getting by storing a hash of the password is that if the stored copy is leaked to an attacker, the attacker doesn't know the passwords themselves. But it's not quite enough just to store something different from the password. We also want to ensure that whatever we store offers an attacker no help in guessing what the password is. If an attacker steals the hashed password, he or she should not be able to analyze the hash to get any clues about the password itself. There is a special class of hashing algorithms called **cryptographic hashes** that make it infeasible to use the hash to figure out what the password is, other than by actually passing a guess at the password through the hashing algorithm. One unfortunate characteris-

tic of cryptographic hashes is that they're hard to design, so even smart people shouldn't try. They use ones created by experts. That's what modern systems should do with password hashing: use a cryptographic hash that has been thoroughly studied and has no known flaws. At any given time, which cryptographic hashing algorithms meet those requirements may vary. At the time of this writing, SHA-3 [B+09] is the US standard for cryptographic hash algorithms, and is a good choice.

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. 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 wrong, if a second guess is allowed, the attacker 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, sooner or later the guess will be 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 type and 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 guessing is called a **dictionary attack**, and it can be highly effective. The dictionary here isn't Websters (or even the Oxford English Dictionary), 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].

If you're smart in setting up your system, an attacker really should not be able to run a dictionary attack on a login process remotely. With any care at all, the attacker will not guess a user's password in the first five or

ASIDE: PASSWORD VAULTS

One way you can avoid the problem of choosing passwords is to use what's called a password vault or key chain. This is an encrypted file kept on your computer that stores passwords. It's encrypted with a password of its own. To get passwords out of the vault, you must provide the password for the vault, reducing the problem of remembering a different password for every site to remembering one password. Also, it ensures that attackers can only use your passwords if they not only have the special password that opens the vault, but they have access to the vault itself. Of course, the benefits of securely storing passwords this way are limited to the strength of the passwords stored in the vault, since guessing and dictionary attacks will still work. Some password vaults will generate strong passwords for you – not very memorable ones, but that doesn't matter, since it's the vault that needs to remember it, not you. You can also find password vaults that store your passwords in the cloud. If you provide them with cleartext versions of your password to store them, however, you are sharing a password with another entity that doesn't really need to know it, thus taking a risk that perhaps you shouldn't take. If the cloud stores only your encrypted passwords, the risk is much lower.

six guesses (alas, sometimes no care is taken and the attacker will), and there's no good reason your system should allow a remote user to make 15,000 guesses at an account's password without getting it right. So by either shutting off access to an account when too many wrong guesses are made at its password, or (better) by drastically slowing down the process of password checking after a few wrong guesses (which makes a long dictionary attack take an infeasible amount of time), you can protect the account against such attacks.

But what if the attacker stole your password file? Since we assume you've been paying attention, it contains hashes of passwords, not passwords itself. But we also assume you paid attention when we told you to use a widely known cryptographic hash, and if you know it, so does the person who stole your password file. If the attacker obtained your hashed passwords, the hashing algorithm, a dictionary, and some compute power, the attacker can crank away at guessing your passwords at their leisure. Worse, if everyone used the same cryptographic hashing algorithm (which, in practice, they probably will), the attacker only needs to run each possible password through the hash once and store the results (essentially, the dictionary has been translated into hashed form). So when the attacker steals your password file, he or she would just need to do string comparisons to your hashed passwords and the newly created dictionary of hashed passwords, which is much faster.

There's a simple fix: before hashing a new password and storing it in your password file, generate a big random number (say 32 or 64 bits) and concatenate it to the password. Hash the result and store that. You also need to store that random number, since when the user tries to log

in and provides the correct password, you'll need to take what the user provided, concatenate the stored random number, and run that through the hashing algorithm. Otherwise, the password hashed by itself won't match what you stored. You typically store the random number (which is called a **salt**) in the password file right next to the hashed password. This concept was introduced in Robert Morris and Ken Thompson's early paper on password security [MT79].

Why does this help? The attacker can no longer create one translation of passwords in the dictionary to their hashes. What is needed is one translation for every possible salt, since the password files that were stolen are likely to have a different salt for every password. If the salt is 32 bits, that's 2^{32} different translations for each word in the dictionary, which makes the approach of pre-computing the translations infeasible. Instead, for each entry in the stolen password file, the dictionary attack must freshly hash each guess with the password's salt. The attack is still feasible if you have chosen passwords badly, but it's not nearly as cheap. Any good system that uses passwords and cares about security stores cryptographically hashed and salted passwords. If yours doesn't, you're putting your users at risk.

There are other troubling issues for the use of passwords, but many of those are not particular to the OS, 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 (PIN). That's essentially a password. But you also need to provide further evidence of your identity...

54.5 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 has, 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?

ASIDE: LINUX LOGIN PROCEDURES

Linux, in the tradition of earlier Unix systems, authenticates users based on passwords and then ties that identity to an initial process associated with the newly logged in user, much as described above. Here we will provide a more detailed step-by-step description of what actually goes on when a user steps up to a keyboard and tries to log in to a Unix system, as a solid example of how a real operating system handles this vital security issue.

1. A special login process running under a privileged system identity displays a prompt asking for the user to type in his or her identity, in the form of a generally short user name. The user types in a user name and hits carriage return. The name is echoed to the terminal.
2. The login process prompts for the user's password. The user types in the password, which is not echoed.
3. The login process looks up the name the user provided in the password file. If it is not found, the login process rejects the login attempt. If it is found, the login process determines the internal user identifier (a unique user ID number), the group (another unique ID number) that the user belongs to, the initial command shell that should be provided to this user once login is complete, and the home directory that shell should be started in. Also, the login process finds the salt and the salted, hashed version of the correct password for this user, which are permanently stored in a secure place in the system.
4. The login process combines the salt for the user's password and the password provided by the user and performs the hash on the combination. It compares the result to the stored version obtained in the previous step. If they do not match, the login process rejects the login attempt.
5. If they do match, fork a process. Set the user and group of the forked process to the values determined earlier, which the privileged identity of the login process is permitted to do. Change directory to the user's home directory and exec the shell process associated with this user (both the directory name and the type of shell were determined in step 3).

There are some other details associated with ensuring that we can log in another user on the same terminal after this one logs out that we don't go into here.

Note that in steps 3 and 4, login can fail either because the user name is not present in the system or because the password does not match the user name. Linux and most other systems do not indicate which condition failed, if one of them did. This choice prevents attackers from learning the names of legitimate users of the system just by typing in guesses, since they cannot know if they guessed a non-existent name or guessed the wrong password for a legitimate user name. Not providing useful information to non-authenticated users is generally a good security idea that has applicability in other types of systems.

Think a bit about why Linux's login procedure chooses to echo the typed user name when it doesn't echo the password. Is there no security disadvantage to echoing the user name, is it absolutely necessary to echo the user name, or is it a tradeoff of security for convenience? Why not echo the password?

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 (sometimes called **dongles**, an unfortunate choice of name) 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 he or she has to the system where the authentication is required. 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 was supposed to be typed 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 or herself. 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 your smartphone on your dresser bureau this morning? What if your dongle 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 your secret authentication device? 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, the thief will still have to guess that before they can pose as you[4].

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

[4] Assuming, of course, you haven't written the password with a Sharpie onto the back of the smart card the thief stole. Well, it seemed like a good idea at the time...

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 a message into the computer. Thinking about this in theory, it 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 the attacker 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[5]. Keep this lesson in mind. Even if it isn't on the test[6], it may come in handy some time in your later career.

54.6 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 with 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 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 much the same extent as the other authentication mechanisms. We can use it, but it won't be perfect, and has its own set of problems and challenges.

[5]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.

[6]We don't know about you, but every time the word "test" or "quiz" or "exam" comes up, our heart skips a beat or two. Too many years of being a student will do this to a person.

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 determining if it belongs to a particular person. Think about what that entails. 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.

You should have identified a few challenges here. 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 that they are the actual owner. What if it's someone who looks a lot like the right user, but isn't? What if the person is wearing a mask? What if the person holds up a photo of the right user, instead of their own face? What if the lighting is dim, or the person isn't fully facing the camera? Alternately, what if it is the right user and the person is not facing the camera, or the lighting is dim, or something else has changed about the person's look? (e.g., hairstyle)

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 using some algorithm. So that "photo" you took is actually a collection of numbers, indicating shadow and light, shades of color, contrasts, and the like. OK, now what? Time to decide if it's the right person's photo or not! How?

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 (also in the form of zeros and ones) 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. Generally this will involve extracting 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.

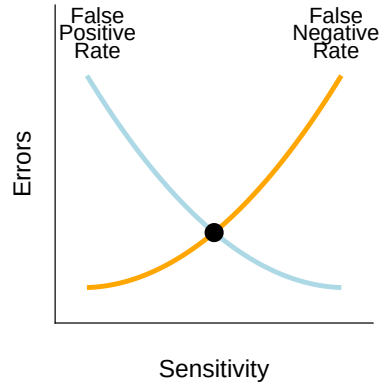

Figure 54.1: **Crossover Error Rate**

Even here, though, an exact match is not too likely. The lighting, for example, might slightly alter the perceived eye color. 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 will look for close matches, not perfect 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 measured versus stored data, we will authenticate a user whom is not who they claim to be. That's 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 usually can't minimize both. As the false positive rate goes down, the false negative rate goes up, and vice versa. The **sensitivity** describes how close the match must be.

Figure 54.1 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, requiring the bank manager to occasionally provide a second scan isn't too bad, while allowing a robber to open the vault with a bogus fake eye would be a disaster. Low false positives might be better here.

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. While a few biometric techniques (such as measuring typing patterns) require relatively common hardware that is likely to be present on many machines anyway, there aren't many such techniques. Even if a special hardware device is available, the convenience of using them for this purpose can be limiting.

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 are bits, and anyone can create any bit pattern they want. If a remote 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 your system and its environment. A good fingerprint reader built in to a smart phone might do its job quite well. 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.

54.7 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 go-between (here, the login) and provide some mechanism whereby the privileged user is permitted to create processes that belongs not to that user, 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, if you type the following:

```
sudo -u webserver apache2
```

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 some employee behind that counter, perhaps giving the person a bunch of personal information, maybe even 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 the person; you weren't shown an official ID card; the person didn't recite the secret DMV mantra that proved he or she was an initiate of that agency. You believed it because the person was standing behind a particular counter, which is the counter DMV employees stand behind. You authenticated the person based on location.

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. (Google introduced multi-factor authentication of this kind in its Android phones, for example.) 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.

This would indicate that the apache2 program should be started under the identity of `webserver`, rather than under the identity of whoever ran the sudo command. This command might require the user running it to provide their own authentication credentials (for extra certainty that it really is the privileged user asking for it, and not some random visitor accessing the 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`. We'll say more about sudo in the chapter on access control.

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. This kind of group is another example of a security-relevant principal, since we will make our decisions on the basis of group membership, rather than individual identity. When one of the system administrators wished to do something requiring group membership, we would check that he or she was a member. We can either associate a group membership with each process, or use the process's 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.

54.8 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 their 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 the user has (like a smart card or proof of possession of a smart phone), or something the user is (like the user's 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 of more than one form, such as requiring both a password and a one-time code that was texted to the user's smart phone.

References

[B+09] "The road from Panama to Keccak via RadioGatun" by Guido Bertoni, Joan Daemen, Michael Peeters, Gilles Van Assche. *The authors who developed SHA-3. For a more readable version, try the Wikipedia page first about SHA-3. There, you learn about the "sponge construction", which actually has something to do with cryptographic hashes, and not the cleaning of your kitchen.*

[C100] "Letter of recommendation to Tiberius Claudius Hermeros" by Celer the Architect. Circa 100 A.D.. *This letter introduced a slave to the imperial procurator, thus providing said procurator evidence that the slave was who he claimed to be. Read the translation at the following website* `http://papyri.info/ddbdp/c.ep.lat;;81`.

[G13] "Anatomy of a hack: even your 'complicated' password is easy to crack" by 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.*

[JB-500] "Judges 12, verses 5-6" The Bible, roughly 5th century BC. *An early example of the use of biometrics. Failing this authentication had severe consequences, as the Gileadites slew mispronouncers, some 42,000 of them according to the book of Judges.*

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

[MT79] "Password Security: A Case History" by 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. And it was the first paper to discuss the technique of password salting.*

[M+02] "Impact of Artificial "Gummy" Fingers on Fingerprint Systems" by 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.*

[P-46] "The Histories" by 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" by 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.*

55

Access Control

Chapter by **Peter Reiher (UCLA)**

55.1 Introduction

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

There are two important steps here:

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

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

open("/var/foo", O_RDWR)

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

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

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

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

55.2 Important Aspects Of The Access Control Problem

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

55.3 Using ACLs For Access Control

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

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

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

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

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

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

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

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

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

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

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

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

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

ASIDE: NAME SPACES

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

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

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

55.4 Using Capabilities For Access Control

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

55.5 Mandatory And Discretionary Access Control

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

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

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

55.6 Practicalities Of Access Control Mechanisms

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

ASIDE: THE ANDROID ACCESS CONTROL MODEL

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

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

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

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

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

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

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

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

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

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

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

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

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

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

TIP: PRIVILEGE ESCALATION CONSIDERED DANGEROUS

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

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

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

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

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

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

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

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

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

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

55.7 Summary

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

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

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

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

References

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

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

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

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

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

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

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

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

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

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

56

Protecting Information With Cryptography

Chapter by **Peter Reiher (UCLA)**

56.1 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 of which it is given control. 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 it is obtained, our secrets are safe. Further, if the attacker cannot understand it, it probably can't be altered, at least not in a controllable way. If the attacker doesn't know what the data means, how can it be changed into something the attacker prefers?

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 opponents from altering the data to suit their desires, and even know when opponents have tampered with our data. All through the joys of cryptography!

But using cryptography properly is not easy, and many uses of cryptography are computationally expensive. So we need to be selective about where and when we use cryptography, and careful in how we implement it and integrate it into our systems. Well chosen uses that are properly performed will tremendously increase security. Poorly chosen uses that are badly implemented won't help at all, and may even hurt.

THE CRUX OF THE PROBLEM:
HOW TO PROTECT INFORMATION OUTSIDE THE OS'S DOMAIN

How can we use cryptography to ensure that, even if others gain access to critical data outside the control of the operating system, they will be unable to either use or alter it? What cryptographic 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?

56.2 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.

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

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 (which is called a **key**), 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 key K, and an encryption algorithm $E()$. We end up with C, the altered form of P, which we usually call the **ciphertext**:

$$C = E(P, K) \tag{56.1}$$

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 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:

$$P = D(C, K) \tag{56.2}$$

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 they still won't be able to learn what P is.

Provided, of course, that ...

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 they are known, the opponent will 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, it can't be understood. But what if the opponent can alter it? What's being altered is the encrypted form, i.e., making 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 the 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 with 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.

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

TIP: DEVELOPING YOUR OWN CIPHERS: DON'T DO IT

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. They might ignore it if you just go around bragging about your wonderful cipher (since they hear that all the time), but if you actually use it for something important, you will unfortunately draw their attention. 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, which is easier to do, anyway.

So, don't.

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.

The form of cryptography we just described 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.

56.3 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 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

$$C = E(P, K_{encrypt}) \tag{56.3}$$

And our decryption operation becomes

$$P = D(C, K_{decrypt}) \tag{56.4}$$

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 publicly 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 things it's 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, messages created by Eve would be misidentified by Bob as originating from Alice, subverting our entire goal of authenticating the messages. We'd better make sure Eve can't fool Bob about which public key belongs to Alice.

This leads down a long and 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 discuss how, in practice, we distribute public keys in a 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?

56.4 Cryptographic Hashes

As we discussed earlier, we can protect data integrity by using cryptography, since alterations to encrypted data will not decrypt properly. We can reduce the costs of that integrity check by hashing the data and encrypting just the hash, instead of encrypting the entire thing. However, if we want to be really careful, we can't use just any hash function, since hash functions, by their very nature, have **hash collisions**, where two different bit patterns hash to the same thing. If an attacker can change the bit pattern we intended to send to some other bit pattern that hashes to the same thing, we would lose our integrity property.

So to be particularly careful, we can use a **cryptographic hash** to ensure integrity. Cryptographic hashes are a special category of hash functions with several important properties:

- It is computationally infeasible to find two inputs that will produce the same hash value.
- Any change to an input will result in an unpredictable change to the resulting hash value.
- It is computationally infeasible to infer any properties of the input based only on the hash value.

Based on these properties, if we only care about data integrity, rather than secrecy, we can take the cryptographic hash of a piece of data, encrypt only that hash, and send both the encrypted hash and the unencrypted 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].

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

To formalize it a bit, to perform a cryptographic hash we take a plaintext P and a hashing algorithm $H()$. Note that there is not necessarily any key involved. Here's what happens:

$$S = H(P) \tag{56.5}$$

Since cryptographic hashes are a subclass of hashes in general, we normally expect S to be shorter than P, perhaps a lot shorter. That implies there will be collisions, situations in which two different plaintexts P and P' both hash to S. However, the properties of cryptographic hashes outlined above will make it difficult for an adversary to make use of collisions. Even if you know both S and P, it should be hard to find any other plaintext P' that hashes to S[3]. It won't be hard to figure out what S' should be for an altered value of plaintext P', since you can simply apply the cryptographic hashing algorithm directly to P'. But even a slightly altered version of P, such as a P' differing only in one bit, should produce a hash S' that differs from S in completely unpredictable ways.

Cryptographic hashes can be used for other purposes than ensuring integrity of encrypted data, as well. They are the class of hashes of choice for storing salted hashed passwords, for example, as discussed in the chapter on authentication. They can be used to determine if a stored file has been altered, a function provided by well-known security software like Tripwire. They can also be used to force a process to perform a certain amount of work before submitting a request, an approach called "proof of work." The submitter is required to submit a request that hashes to a certain value using some specified cryptographic hash, which, because of the properties of such hashes, requires them to try a lot of request formats before finding one that hashes to the required value. Since each hash operation takes some time, submitting a proper request will require a predictable amount of work. This use of hashes, in varying forms, occurs in several applications, including spam prevention and blockchains.

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. However, there is a history of cryptographic hashing algorithms becoming obsolete, so if you are designing a system that uses one, it's wise to first check to see what current recommendations are for choices of such an algorithm.

56.5 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?

[3]Every so often, a well known cryptographic hashing function is "broken" in the sense that someone figures out how to create a P' that uses the function to produce the same hash as P. That happened to a hashing function known as SHA-1 in 2017, rendering that function unsafe and unusable for integrity purposes [G17].

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]. Examples include distributing secret keys in software shared by many people, incorrectly transmitting plaintext versions of keys across a network, and choosing keys from a seriously reduced set of possible choices, rather than the larger theoretically possible set. 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, nor in its key selection procedures. This flaw allowed attackers to read the traffic of something between $1/4$ and $1/2$ 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

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. This highly desirable property in a cryptographic system is called **perfect forward secrecy**.

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, like low order bits of the times required for pieces of hardware to perform operations, 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`. Windows has a similar entropy-gathering feature. Use these to generate your keys. They're not perfect, but they're good enough for many purposes.

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[4].

As another example, an early implementation of the Netscape web browser generated cryptographic keys using some easily guess-able 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

[4]WEP got replaced by WPA. Unfortunately, WPA proved to have its own weaknesses, so it was replaced by WPA2. Unfortunately, WPA2 proved to have its own weaknesses, so it is being replaced by WPA3, as of 2018. The sad fate of providing cryptography for wireless networks should serve as a lesson to any of you tempted to underestimate the difficulties in getting this stuff right.

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 learn 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.

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. Or, as you might have guessed: **THE CRYPTOGRAPHY'S BENEFIT RELIES ENTIRELY ON THE SECRECY OF THE KEY.**

56.6 Cryptography And Operating Systems

Cryptography is fascinating, but lots of things are fascinating[5], 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[6].

[5] For example, the late piano Sonatas of Beethoven. One movement of his last Sonata, Opus 111, even sounds like jazz, while being written in the 1820s!

[6] But remember our discussion of security enclaves in an earlier chapter, hardware that does not allow the operating system full access to information that the enclave protects. Think for a moment what the implications of that are for cryptography on a computer using such an enclave, and what new possibilities it offers.

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 them your data in plaintext and asking them to encrypt it. They've seen the plaintext and can keep a copy of the key.

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

One can argue that not all compromises of an operating system are permanent. Many are, but some only give an attacker temporary access to system resources, or perhaps access to only a few particular resources. In such cases, if the encrypted data is not stored in plaintext and the decryption key is not available at the time or in the place the attacker can access, encrypting that data may still provide benefit. The tricky issue here is that you can't know ahead of time whether successful attacks on your system will only occur at particular times, for particular durations, or on particular elements of the system. So if you take this approach, you want to minimize all your exposure: decrypt infrequently, dispose of plaintext data quickly and carefully, and don't keep a plaintext version of the key in the system except when performing the cryptographic operations. Such minimization can be difficult to achieve.

If cryptography won't protect us completely against a dishonest operating system, what OS 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 OS is compromised, since the encrypted passwords can't be decrypted[7].

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

[7] But if the legitimate user ever provides the correct password to a compromised OS, all bets are off, alas. The compromised OS 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.

final destination machine will need the key if he or 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.

56.7 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[8]. 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

[8] 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.

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. For example, it offers little protection against buffer overflows or SQL injections.
- It does not protect against dishonest privileged users on the system, such as a system administrator. Administrator's privileges may allow the admin to pose as the user who owns the data or to install system components that provide access to the user's data; thus, the admin could access decrypted copies of the data on request.
- It does not protect against security flaws in the OS 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 OS 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. Note, however, that if the key is lost, you will never be able to decrypt the archived data.
- 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 – remember, when you trust another computing element, you're trusting not only its honesty, but also its carefulness and correctness), 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**), which we discussed in the

authentication chapter. 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 required 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 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 a password was needed (remember the principle of acceptability?). 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 a password when the system is first accessed, but is not required to re-authenticate again until logging 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. Both have the tremendous advantage that they don't annoy their users so much that they are abandoned in favor of systems offering no security whatsoever.

56.8 Cryptographic Capabilities

Remember from our chapter on access control that capabilities had the problem that we could not leave them in users' hands, since then users could forge them and grant themselves access to anything they wanted. Cryptography can be used to create unforgeable capabilities. A trusted entity could use cryptography to create a sufficiently long and securely encrypted data structure that indicated that the possessor was allowed to have access to a particular resource. This data structure could then be given to a user, who would present it to the owner of the matching resource to obtain access. The system that actually controlled the resource must be able to check the validity of the data structure before granting access, but would not need to maintain an access control list.

Such cryptographic capabilities could be created either with symmetric or public key cryptography. With symmetric cryptography, both the creator of the capability and the system checking it would need to share the same key. This option is most feasible when both of those entities are the same system, since otherwise it requires moving keys around between the machines that need to use the keys, possibly at high speed and scale, depending on the use scenario. One might wonder why the single machine would bother creating a cryptographic capability to allow access, rather than simply remembering that the user had passed an access check, but there are several possible reasons. For example, if the machine controlling the resource worked with vast numbers of users, keeping track of the access status for each of them would be costly and complex, particularly in a distributed environment where the system needed to worry about failures and delays. Or if the system wished to give transferable rights to the access, as it might if the principal might move from machine to machine, it would be more feasible to allow the capability to move with the principal and be used from any location. Symmetric cryptographic capabilities also make sense when all of the machines creating and checking them are inherently trusted and key distribution is not problematic.

If public key cryptography is used to create the capabilities, then the creator and the resource controller need not be co-located and the trust relationships need not be as strong. The creator of the capability needs one key (typically the secret key) and the controller of the resource needs the other. If the content of the capability is not itself secret, then a true public key can be used, with no concern over who knows it. If secrecy (or at least some degree of obscurity) is required, what would otherwise be a public key can be distributed only to the limited set of entities that would need to check the capabilities[9]. A resource manager could create a set of credentials (indicating which principal was allowed to use what resources, in what ways, for what period of time) and then encrypt them with a private key. Any one else can validate those credentials by decrypting them with the manager's public key. As long as only the resource manager knows the private key, no one can forge capabilities.

As suggested above, such cryptographic capabilities can hold a good deal of information, including expiration times, identity of the party who was given the capability, and much else. Since strong cryptography will ensure integrity of all such information, the capability can be relied upon. This feature allows the creator of the capability to prevent arbitrary copying and sharing of the capability, at least to a certain extent. For example, a cryptographic capability used in a network context can be tied to a particular IP address, and would only be regarded as valid if the message carrying it came from that address.

[9]Remember, however, that if you are embedding a key in a piece of widely distributed software, you can count on that key becoming public knowledge. So even if you believe the matching key is secret, not public, it is unwise to rely too heavily on that belief.

Many different encryption schemes can be used. The important point is that the encrypted capabilities must be long enough that it is computationally infeasible to find a valid capability by brute force enumeration or random guessing (e.g., the number of invalid bit patterns is 10^{15} times larger than the number of valid bit patterns).

We'll say a bit more about cryptographic capabilities in the chapter on distributed system security.

56.9 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 use 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. Cryptographic hashes, on the other hand, do not allow reversal of the cryptography and do not require the use of keys.

Strong ciphers make it computationally infeasible to obtain the original bit pattern without access to the required key. For symmetric and asymmetric ciphers, this implies that only holders of the proper key can obtain the cipher's benefits. Since cryptographic hashes have no key, this implies that no one should be able to obtain the original bit pattern from the hash.

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. A good principle is to store keys in as few places as possible, for as short a duration as possible, available to as few parties as possible.

References

[D88] "The First Ten Years of Public Key Cryptography" by 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" by 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" by 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.*

[G17] "SHA-1 Shattered" by Google. `https://shattered.io`, 2017. *A web site describing details of how Google demonstrated the insecurity of the SHA-1 cryptographic hashing function. The web site provides general details, but also includes a link to a technical paper describing exactly how it was done.*

[GW96] "Randomness and the Netscape Browser" by 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? Don't you wish they weren't?*

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

[S96] "Applied Cryptography" by 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" by Stefan Viehbock. Available on: `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.*

57

Distributed System Security

Chapter by **Peter Reiher (UCLA)**

57.1 Introduction

An operating system can only control its own machine's resources. Thus, operating systems will have challenges in providing security in distributed systems, where more than one machine must cooperate. There are two large problems:

- The other machines in the distributed system might not properly implement the security policies you want, or they might be adversaries impersonating trusted partners. We cannot control remote systems, but we still have to be able to trust validity of the credentials and capabilities they give us.
- Machines in a distributed system communicate across a network that none of them fully control and that, generally, cannot be trusted. Adversaries often have equal access to that network and can forge, copy, replay, alter, destroy, and delay our messages, and generally interfere with our attempts to use the network.

As suggested earlier, cryptography will be the major tool we use here, but we also said cryptography was hard to get right. That makes it sound like the perfect place to use carefully designed standard tools, rather than to expect everyone to build their own. That's precisely correct. As such:

THE CRUX: HOW TO PROTECT DISTRIBUTED SYSTEM OPERATIONS
How can we secure a system spanning more than one machine? What tools are available to help us protect such systems? How do we use them properly? What are the areas in using the tools that require us to be careful and thoughtful?

57.2 The Role of Authentication

How can we handle our uncertainty about whether our partners in a distributed system are going to enforce our security policies? In most cases, we can't do much. At best, we can try to arrange to agree on policies and hope everyone follows through on those agreements. There are some special cases where we can get high-quality evidence that our partners have behaved properly, but that's not easy, in general. For example, how can we know that they are using full disk encryption, or that they have carefully wiped an encryption key we are finished using, or that they have set access controls on the local copies of their files properly? They can say they did, but how can we *know*?

Generally, we can't. But you're used to that. In the real world, your friends and relatives know some secrets about you, and they might have keys to get into your home, and if you loan them your car you're fairly sure you'll get it back. That's not so much because you have perfect mechanisms to prevent those trusted parties from behaving badly, but because you are pretty sure they won't. If you're wrong, perhaps you can detect that they haven't behaved well and take compensating actions (like changing your locks or calling the police to report your car stolen). We'll need to rely on similar approaches in distributed computer systems. We will simply have to trust that some parties will behave well. In some cases, we can detect when they don't and adjust our trust in the parties accordingly, and maybe take other compensating actions.

Of course, in the cyber world, our actions are at a distance over a network, and all we see are bits going out and coming in on the network. For a trust-based solution to work, we have to be quite sure that the bits we send out can be verified by our buddies as truly coming from us, and we have to be sure that the bits coming in really were created by them. That's a job for authentication. As suggested in the earlier authentication chapter, when working over a network, we need to authenticate based on a bundle of bits. Most commonly, we use a form of authentication based on what you know. Now, think back to the earlier chapters. What might someone running on a remote operating system know that no one else knows? How about a password? How about a private key?

Most of our distributed system authentication will rely on one of these two elements. Either you require the remote machine to provide you with a password, or you require it to provide evidence using a private key stored only on that machine[1]. In each case, you need to know something to check the authentication: either the password (or, better, a cryptographic hash of the password plus a salt) or the public key.

[1] We occasionally use other methods, such as smart cards or remote biometric readers. They are less common in today's systems, though. If you understand how we use passwords and public key cryptography for distributed system authentication, you can probably figure out how to make proper use of these other techniques, too. If you don't, you'll be better off figuring out the common techniques before moving to the less common ones.

When is each appropriate? Passwords tend to be useful if there are a vast number of parties who need to authenticate themselves to one party. Public keys tend to be useful if there's one party who needs to authenticate himself to a vast number of parties. Why? With a password, the authentication provides evidence that somebody knows a password. If you want to know exactly who that is (which is usually important), only the party authenticating and the party checking can know it. With a public key, many parties can know the key, but only one party who knows the matching private key can authenticate himself. So we tend to use both mechanisms, but for different cases. When a web site authenticates itself to a user, it's done with PK cryptography. By distributing one single public key (to vast numbers of users), the web site can be authenticated by all its users. The web site need not bother keeping separate authentication information to authenticate itself to each user. When that user authenticates itself to the web site, it's done with a password. Each user must be separately authenticated to the web site, so we require a unique piece of identifying information for that user, preferably something that's easy for a person to use. Setting up and distributing public keys is hard, while setting up individual passwords is relatively easy.

How, practically, do we use each of these authentication mechanisms in a distributed system? If we want a remote partner to authenticate itself via passwords, we will require it to provide us with that password, which we will check. We'll need to encrypt the transport of the password across the network if we do that; otherwise anyone eavesdropping on the network (which is easy for many wireless networks) will readily learn passwords sent unencrypted. Encrypting the password will require that we already have either a shared symmetric key or our partner's public key. Let's concentrate now on how we get that public key, either to use it directly or set up the cryptography to protect the password in transit.

We'll spend the rest of the chapter on securing the network connection, but please don't forget that even if you secure the network perfectly, you still face the major security challenge of the uncontrolled site you're interacting with on the other side of the network. If your compromised partner attacks you, it will offer little consolation that the attack was authenticated and encrypted.

57.3 Public Key Authentication For Distributed Systems

The public key doesn't need to be secret, but we need to be sure it really belongs to our partner. If we have a face-to-face meeting, our partner can directly give us a public key in some form or another, in which case we can be pretty sure it's the right one. That's limiting, though, since we often interact with partners whom we never see face to face. For that matter, whose "face" belongs to Amazon[2] or Google?

[2]How successful would Amazon be if Jeff Bezos had to make an in-person visit to every customer to deliver them Amazon's public key? Answer: Not as successful.

Fortunately, we can use the fact that secrecy isn't required to simply create a bunch of bits containing the public key. Anyone who gets a copy of the bits has the key. But how do they know for sure whose key it is? What if some other trusted party known to everyone who needs to authenticate our partner used their own public key to cryptographically sign that bunch of bits, verifying that they do indeed belong to our partner? If we could check that signature, we could then be sure that bunch of bits really does represent our partner's public key, at least to the extent that we trust that third party who did the signature.

This technique is how we actually authenticate web sites and many other entities on the Internet. Every time you browse the web or perform any other web-based activity, you use it. The signed bundle of bits is called a **certificate**. Essentially, it contains information about the party that owns the public key, the public key itself, and other information, such as an expiration date. The entire set of information, including the public key, is run through a cryptographic hash, and the result is encrypted with the trusted third party's private key, digitally signing the certificate. If you obtain a copy of the certificate, and can check the signature, you can learn someone else's public key, even if you have never met or had any direct interaction with them. In certain ways, it's a beautiful technology that empowers the whole Internet.

Let's briefly go through an example, to solidify the concepts. Let's say Frobazz Inc. wants to obtain a certificate for its public key, which is KF. Frobazz Inc. pays big bucks to Acmesign Co., a widely trusted company whose business it is to sell certificates, to obtain a certificate signed by AcmeSign. Such companies are commonly called **Certificate Authorities**, or **CAs**, since they create authoritative certificates trusted by many parties. Acmesign checks up on Frobazz Inc. to ensure that the people asking for the certificate actually are legitimate representatives of Frobazz. Acmesign then makes very, very sure that the public key it's about to embed in a certificate actually is the one that Frobazz wants to use. Assuming it is, Acmesign runs a cryptographic hashing algorithm (perhaps SHA-3 which, unlike SHA-1, has not been cracked, as of 2020) on Frobazz's name, public key KF, and other information, producing hash HF. Acmesign then encrypts HF with its own private key, PA, producing digital signature SF. Finally, Acmesign combines all the information used to produce HF, plus Acmesign's own identity and the signature SF, into the certificate CF, which it hands over to Frobazz, presumably in exchange for money. Remember, CF is just some bits.

Now Frobazz Inc. wants to authenticate itself over the Internet to one of its customers. If the customer already has Frobazz's public key, we can use public key authentication mechanisms directly. If the customer does not have the public key, Frobazz sends CF to the customer. The customer examines the certificate, sees that it was generated by Acmesign using, say, SHA-3, and runs the same information that Acmesign hashed (all of which is in the certificate itself) through SHA-3, producing HF'. Then the customer uses Acmesign's public key to decrypt SF (also in the

certificate), obtaining HF. If all is well, HF equals HF', and now the customer knows that the public key in the certificate is indeed Frobazz's. Public key-based authentication can proceed[3]. If the two hashes aren't exactly the same, the customer knows that something fishy is going on and will not accept the certificate.

There are some wonderful properties about this approach to learning public keys. First, note that the signing authority (Acmesign, in our example) did not need to participate in the process of the customer checking the certificate. In fact, Frobazz didn't really, either. The customer can get the certificate from literally anywhere and obtain the same degree of assurance of its validity. Second, it only needs to be done once per customer. After obtaining the certificate and checking it, the customer has the public key that is needed. From that point onward, the customer can simply store it and use it. If, for whatever reason, it gets lost, the customer can either extract it again from the certificate (if that has been saved), or go through the process of obtaining the certificate again. Third, the customer had no need to trust the party claiming to be Frobazz until that identity had been proven by checking the certificate. The customer can proceed with caution until the certificate checks out.

Assuming you've been paying attention for the last few chapters, you should be saying to yourself, "now, wait a minute, isn't there a chicken-and-egg problem here?" We'll learn Frobazz's public key by getting a certificate for it. The certificate will be signed by Acmesign. We'll check the signature by knowing Acmesign's public key. But where did we get Acmesign's key? We really hope you did have that head-scratching moment and asked yourself that question, because if you did, you understand the true nature of the Internet authentication problem. Ultimately, we've got to bootstrap it. You've got to somehow or other obtain a public key for somebody that you trust. Once you do, if it's the right public key for the right kind of party, you can then obtain a lot of other public keys. But without something to start from, you can't do much of anything.

Where do you get that primal public key? Most commonly, it comes in a piece of software you obtain and install. The one you use most often is probably your browser, which typically comes with the public keys for several hundred trusted authorities[4]. Whenever you go to a new web site that cares about security, it provides you with a certificate containing that site's public key, and signed by one of those trusted authorities preconfigured into your browser. You use the pre-configured public key of that authority to verify that the certificate is indeed proper, after which you know the public key of that web site. From that point onward, you can use the web site's public key to authenticate it. There are some se-

[3] And, indeed, must, since all this business with checking the certificate merely told the customer what Frobazz's public key was. It did nothing to assure the customer that whoever sent the certificate actually was Frobazz or knew Frobazz's private key.

[4] You do know of several hundred companies out there that you trust with everything you do on the web, don't you? Well, know of them or not, you effectively trust them to that extent.

rious caveats here (and some interesting approaches to addressing those caveats), but let's put those aside for the moment.

Anyone can create a certificate, not just those trusted CAs, either by getting one from someone whose business it is to issue certificates or simply by creating one from scratch, following a certificate standard (X.509 is the most commonly used certificate standard [I12]). The necessary requirement: the party being authenticated and the parties performing the authentication must all trust whoever created the certificate. If they don't trust that party, why would they believe the certificate is correct?

If you are building your own distributed system, you can create your own certificates from a machine you (and other participants in the system) trust and can handle the bootstrapping issue by carefully hand-installing the certificate signing machine's public key wherever it needs to be. There are a number of existing software packages for creating certificates, and, as usual with critical cryptographic software, you're better off using an existing, trusted implementation rather than coding up one of your own. One example you might want to look at is PGP (available in both supported commercial versions and compatible but less supported free versions) [P16], but there are others. If you are working with a fixed number of machines and you can distribute the public key by hand in some reasonable way, you can dispense entirely with certificates. Remember, the only point of a PK certificate is to distribute the public key, so if your public keys are already where they need to be, you don't need certificates.

OK, one way or another you've obtained the public key you need to authenticate some remote machine. Now what? Well, anything they send you encrypted with their private key will only decrypt with their public key, so anything that decrypts properly with the public key must have come from them, right? Yes, it must have come from them at some point, but it's possible for an adversary to have made a copy of a legitimate message the site sent at some point in the past and then send it again it at some future date. Depending on exactly what's going on, that could cause trouble, since you may take actions based on that message that the legitimate site did not ask for. So usually we take measures to ensure that we're not being subjected to a **replay attack**. Such measures generally involve ensuring that each encrypted message contains unique information not in any other message. This feature is built in properly to standard cryptographic protocols, so if you follow our advice and use one of those, you will get protection from such replay attacks. If you insist on building your own cryptography, you'll need to learn a good deal more about this issue and will have to apply that knowledge very carefully. Also, public key cryptography is expensive. We want to stop using it as soon as possible, but we also want to continue to get authentication guarantees. We'll see how to do that when we discuss SSL and TLS.

57.4 Password Authentication For Distributed Systems

The other common option to authenticate in distributed systems is to use a password. As noted above, that will work best in situations where only two parties need to deal with any particular password: the party being authenticated and the authenticating party. They make sense when an individual user is authenticating himself to a site that hosts many users, such as when you log in to Amazon. They don't make sense when that site is trying to authenticate itself to an individual user, such as when a web site claiming to be Amazon wants to do business with you. Public key authentication works better there.

How do we properly handle password authentication over the network, when it is a reasonable choice? The password is usually associated with a particular user ID, so the user provides that ID and password to the site requiring authentication. That typically happens over a network, and typically we cannot guarantee that networks provide confidentiality. If our password is divulged to someone else, they'll be able to pose as us, so we must add confidentiality to this cross-network authentication, generally by encrypting at least the password itself (though encrypting everything involved is better). So a typical interchange with Alice trying to authenticate herself to Frobazz Inc.'s web site would involve the site requesting a user ID and password and Alice providing both, but encrypting them before sending them over the network.

The obvious question you should ask is, encrypting them with what key? Well, if Frobazz authenticated itself to Alice using PK, as discussed above, Alice can encrypt her user ID and password with Frobazz's public key. Frobazz Inc., having the matching private key, will be able to check them, but nobody else can read them. In actuality, there are various reasons why this alone would not suffice, including replay attacks, as mentioned above. But we can and do use Frobazz's private key to set up cryptography that will protect Alice's password in transit. We'll discuss the details in the section on SSL/TLS.

We discussed issues of password choice and management in the chapter on authentication, and those all apply in the networking context. Otherwise, there's not that much more to say about how we'll use passwords, other than to note that after the remote site has verified the password, what does it actually know? That the site or user who sent the password knows it, and, to the strength of the password, that site or user is who it claims to be. But what about future messages that come in, supposedly from that site? Remember, anyone can create any message they want, so if all we do is verify that the remote site sent us the right password, all we know is that particular message is authentic. We don't want to have to include the password on every message we send, just as we don't want to use PK to encrypt every message we send. We will use both authentication techniques to establish initial authenticity, then use something else to tie that initial authenticity to subsequent interactions. Let's move right along to SSL/TLS to talk about how we do that.

57.5 SSL/TLS

We saw in an earlier chapter that a standard method of communicating between processes in modern systems is the socket. That's equally true when the processes are on different machines. So a natural way to add cryptographic protection to communications crossing unprotected networks is to add cryptographic features to sockets. That's precisely what **SSL** (the **Secure Socket Layer**) was designed to do, many years ago. Unfortunately, SSL did not get it quite right. That's because it's pretty darn hard to get it right, not because the people who designed and built it were careless. They learned from their mistakes and created a new version of encrypted sockets called **Transport Layer Security** (**TLS**)[5]. You will frequently hear people talk about using SSL. They are usually treating it as a shorthand for SSL/TLS. SSL, formally, is insecure and should never be used for anything. Use TLS. The only exception is that some very old devices might run software that doesn't support TLS. In that case, it's better to use SSL than nothing. We'll adopt the same shorthand as others from here on, since it's ubiquitous.

The concept behind SSL is simple: move encrypted data through an ordinary socket. You set up a socket, set up a special structure to perform whatever cryptography you want, and hook the output of that structure to the input of the socket. You reverse the process on the other end. What's simple in concept is rather laborious in execution, with a number of steps required to achieve the desired result. There are further complications due to the general nature of SSL. The technology is designed to support a variety of cryptographic operations and many different ciphers, as well as multiple methods to perform key exchange and authentication between the sender and receiver.

The process of adding SSL to your program is intricate, requiring the use of particular libraries and a sequence of calls into those libraries to set up a correct SSL connection. We will not go through those operations step by step here, but you will need to learn about them to make proper use of SSL. Their purpose is, for the most part, to allow a wide range of generality both in the cryptographic options SSL supports and the ways you use those options in your program. For example, these setup calls would allow you to create one set of SSL connections using AES-128 and another using AES-256, if that's what you needed to do.

One common requirement for setting up an SSL connection that we will go through in a bit more detail is how to securely distribute whatever cryptographic key you will use for the connection you are setting up. Best cryptographic practice calls for you to use a brand new key to encrypt the bulk of your data for each connection you set up. You will use

[5] Actually, even the first couple of versions of TLS didn't get it quite right. As of 2020, the current version of TLS is 1.3, and that's probably what you should use. TLS 1.3 closed some vulnerabilities that TLS 1.2 is subject to, The history of required changes to SSL/TLS should further reinforce the lesson of how hard it is to use cryptography properly, which in turn should motivate you to foreswear ever trying to roll your own crypto.

public/private keys for authentication many times, but as we discussed earlier, you need to use symmetric cryptography to encrypt the data once you have authenticated your partner, and you want a fresh key for that. Even if you are running multiple simultaneous SSL connections with the same partner, you want a different symmetric key for each connection.

So what do you need to do to set up a new SSL connection? We won't go through all of the gory details, but, in essence, SSL needs to bootstrap a secure connection based (usually) on asymmetric cryptography when no usable symmetric key exists. (You'll hear "usually" and "normally" and "by default" a lot in SSL discussions, because of SSL's ability to support a very wide range of options, most of which are ordinarily not what you want to do.) The very first step is to start a negotiation between the client and the server. Each party might only be able to handle particular ciphers, secure hashes, key distribution strategies, or authentication schemes, based on what version of SSL they have installed, how it's configured, and how the programs that set up the SSL connection on each side were written. In the most common cases, the negotiation will end in both sides finding some acceptable set of ciphers and techniques that hit a balance between security and performance. For example, they might use RSA with 2048 bit keys for asymmetric cryptography, some form of a Diffie-Hellman key exchange mechanism (see the Aside on this mechanism) to establish a new symmetric key, SHA-3 to generate secure hashes for integrity, and AES with 256 bit keys for bulk encryption. A modern installation of SSL might support 50 or more different combinations of these options.

In some cases, it may be important for you to specify which of these many combinations are acceptable for your system, but often most of them will do, in which case you can let SSL figure out which to use for each connection without worrying about it yourself. The negotiation will happen invisibly and SSL will get on with its main business: authenticating at least the server (optionally the client), creating and distributing a new symmetric key, and running the communication through the chosen cipher using that key.

We can use Diffie-Hellman key exchange to create the key (and SSL frequently does), but we need to be sure who we are sharing that key with. SSL offers a number of possibilities for doing so. The most common method is for the client to obtain a certificate containing the server's public key (typically by having the server send it to the client) and to use the public key in that certificate to verify the authenticity of the server's messages. It is possible for the client to obtain the certificate through some other means, though less common. Note that having the server send the certificate is every bit as secure (or insecure) as having the client obtain the certificate through other means. Certificate security is not based on the method used to transport it, but on the cryptography embedded in the certificate.

With the certificate in hand (however the client got it), the Diffie-Hellman key exchange can now proceed in an authenticated fashion. The server

ASIDE: DIFFIE-HELLMAN KEY EXCHANGE

What if you want to share a secret key between two parties, but they can only communicate over an insecure channel, where eavesdroppers can hear anything they say? You might think this is an impossible problem to solve, but you'd be wrong. Two extremely smart cryptographers named Whitfield Diffie and Martin Hellman solved this problem years ago, and their solution is in common use. It's called **Diffie-Hellman key exchange**.

Here's how it works. Let's say Alice and Bob want to share a secret key, but currently don't share anything, other than the ability to send each other messages. First, they agree on two numbers, n (a large prime number) and g (which is primitive mod n). They can use the insecure channel to do this, since n and g don't need to be secret. Alice chooses a large random integer, say x, calculates $X = g^x mod\ n$, and sends X to Bob. Bob independently chooses a large random integer, say y, calculates $Y = g^y mod\ n$, and sends Y to Alice. The eavesdroppers can hear X and Y, but since Alice and Bob didn't send x or y, the eavesdroppers don't know those values. It's important that Alice and Bob keep x and y secret.

Alice now computes $k = Y^x mod\ n$, and Bob computes $k = X^y mod\ n$. Alice and Bob get the same value k from these computations. Why? Well, $Y^x mod\ n = (g^y\ mod\ n)^x\ mod\ n$, which in turn equals $g^{yx}\ mod\ n$. $X^y\ mod\ n = (g^x\ mod\ n)^y\ mod\ n = g^{xy}\ mod\ n$, which is the same thing Alice got. Nothing magic there, that's just how exponentiation and modulus arithmetic work. Ah, the glory of mathematics! So k is the same in both calculations and is known to both Alice and Bob.

What about those eavesdroppers? They know g, n, X, and Y, but not x or y. They can compute $k' = XY\ mod\ n$, but that is not equal to the k Alice and Bob calculated. They do have approaches to derive x or y, which would give them enough information to obtain k, but those approaches require them either to perform a calculation for every possible value of n (which is why you want n to be very large) or to compute a discrete logarithm. Computing a discrete logarithm is a solvable problem, but it's computationally infeasible for large numbers. So if the prime n is large (and meets other properties), the eavesdroppers are out of luck. How large? 600 digit primes should be good enough.

Neat, no? But there is a fly in the ointment, when one considers using Diffie-Hellman over a network. It ensures that you securely share a key with someone, but gives you no assurance of who you're sharing the key with. Maybe Alice is sharing the key with Bob, as she thinks and hopes, but maybe she's sharing it with Mallory, who posed as Bob and injected his own Y. Since we usually care who we're in secure communication with, we typically augment Diffie-Hellman with an authentication mechanism to provide the assurance of our partner's identity.

will sign its Diffie-Hellman messages with its private key, which will allow the client to determine that its partner in this key exchange is the correct server. Typically, the client does not provide (or even have) its own certificate, so it cannot sign its Diffie-Hellman messages. This implies that when SSL's Diffie-Hellman key exchange completes, typically the client is pretty sure who the server is, but the server has no clue about the client's identity. (Again, this need not be the case for all uses of SSL. SSL includes connection creation options where both parties know each other's public key and the key exchange is authenticated on both sides. Those options are simply not the most commonly used ones, and particularly are not the ones typically used to secure web browsing.)

Recalling our discussion earlier in this chapter, it actually isn't a problem for the server to be unsure about the client's identity at this point, in many cases. As we stated earlier, the client will probably want to use a password to authenticate itself, not a public key extracted from a certificate. As long as the server doesn't permit the client to do anything requiring trust before the server obtains and checks the client's password, the server probably doesn't care who the client is, anyway. Many servers offer some services to anonymous clients (such as providing them with publicly available information), so as long as they can get a password from the client before proceeding to more sensitive subjects, there is no security problem. The server can ask the client for a user ID and password later, at any point after the SSL connection is established. Since creating the SSL connection sets up a symmetric key, the exchange of ID and password can be protected with that key.

A final word about SSL/TLS: it's a protocol, not a software package. There are multiple different software packages that implement this protocol. Ideally, if they all implement the protocol properly, they all interact correctly. However, they use different code to implement the protocol. As a result, software flaws in one implementation of SSL/TLS might not be present in other implementations. For example, the Heartbleed attack was based on implementation details of OpenSSL [H14], but was not present in other implementations, such as the version of SSL/TLS found in Microsoft's Windows operating system. It is also possible that the current protocol definition of SSL/TLS contains protocol flaws that would be present in any compliant implementation. If you hear of a security problem involving SSL, determine whether it is a protocol flaw or an implementation flaw before taking further action. If it's an implementation flaw, and you use a different implementation, you might not need to take any action in response.

57.6 Other Authentication Approaches

While passwords and public keys are the most common ways to authenticate a remote user or machines, there are other options. One such option is used all the time. After you have authenticated yourself to a web site by providing a password, as we described above, the web site

will continue to assume that the authentication is valid. It won't ask for your password every time you click a link or perform some other interaction with it. (And a good thing, too. Imagine how much of a pain it would be if you had to provide your password every time you wanted to do anything.) If your session is encrypted at this point, it could regard your proper use of the cryptography as a form of authentication; but you might even be able to quit your web browser, start it up again, navigate back to that web site, and still be treated as an authenticated user, without a new request for your password. At that point, you're no longer using the same cryptography you used before, since you would have established a new session and set up a new cryptographic key. How did your partner authenticate that you were the one receiving the new key?

In such cases, the site you are working with has chosen to make a security tradeoff. It verified your identity at some time in the past using your password and then relies on another method to authenticate you in the future. A common method is to use **web cookies**. Web cookies are pieces of data that a web site sends to a client with the intention that the client stores that data and send it back again whenever the client next communicates with the server. Web cookies are built into most browsers and are handled invisibly, without any user intervention. With proper use of cryptography, a server that has verified the password of a client can create a web cookie that securely stores the client's identity. When the client communicates with the server again, the web browser automatically includes the cookie in the request, which allows the server to verify the client's identity without asking for a password again[6].

If you spend a few minutes thinking about this authentication approach, you might come up with some possible security problems associated with it. The people designing this technology have dealt with some of these problems, like preventing an eavesdropper from simply using a cookie that was copied as it went across the network. However, there are other security problems (like someone other than the legitimate user using the computer that was running the web browser and storing the cookie) that can't be solved with these kinds of cookies, but could have been solved if you required the user to provide the password every time. When you build your own system, you will need to think about these sorts of security tradeoffs yourself. Is it better to make life simpler for your user by not asking for a password except when absolutely necessary, or is it better to provide your user with improved security by frequently requiring proof of identity? The point isn't that there is one correct an-

[6] You might remember from the chapter on access control that we promised to discuss protecting capabilities in a network context using cryptography. That, in essence, is what these web cookies are. After a user authenticates itself with another mechanism, the remote system creates a cryptographic capability for that user that no one else could create, generally using a key known only to that system. That capability/cookie can now be passed back to the other party and used for future authorization operations. The same basic approach is used in a lot of other distributed systems.

swer to this question, but that you need to think about such questions in the design of your system.

There are other authentication options. One example is what is called a **challenge/response protocol**. The remote machine sends you a challenge, typically in the form of a number. To authenticate yourself, you must perform some operation on the challenge that produces a response. This should be an operation that only the authentic party can perform, so it probably relies on the use of a secret that party knows, but no one else does. The secret is applied to the challenge, producing the response, which is sent to the server. The server must be able to verify that the proper response has been provided. A different challenge is sent every time, requiring a different response, so attackers gain no advantage by listening to and copying down old challenges and responses. Thus, the challenges and responses need not be encrypted. Challenge/response systems usually perform some kind of cryptographic operation, perhaps a hashing operation, on the challenge plus the secret to produce the response. Such operations are better performed by machines than people, so either your computer calculates the response for you or you have a special hardware token that takes care of it. Either way, a challenge/response system requires pre-arrangement between the challenging machine and the machine trying to authenticate itself. The hardware token or data secret must have been set up and distributed before the challenge is issued.

Another authentication option is to use an authentication server. In essence, you talk to a server that you trust and that trusts you. The party you wish to authenticate to must also trust the server. The authentication server vouches for your identity in some secure form, usually involving cryptography. The party who needs to authenticate you is able to check the secure information provided by the authentication server and thus determine that the server verified your identity. Since the party you wish to communicate with trusts the authentication server, it now trusts that you are who you claim to be. In a vague sense, certificates and CAs are an offline version of such authentication servers. There are more active online versions that involve network interactions of various sorts between the two machines wishing to communicate and one or more authentication servers. Online versions are more responsive to changes in security conditions than offline versions like CAs. An old certificate that should not be honored is hard to get rid of, but an online authentication server can invalidate authentication for a compromised party instantly and apply the changes immediately. The details of such systems can be quite complex, so we will not discuss them in depth. Kerberos is one example of such an online authentication server [NT94].

57.7 Some Higher Level Tools

In some cases, we can achieve desirable security effects by working at a higher level. **HTTPS** (the cryptographically protected version of the HTTP protocol) and **SSH** (a competitor to SSL most often used to set up secure sessions with remote computers) are two good examples.

HTTPS

HTTP, the protocol that supports the World Wide Web, does not have its own security features. Nowadays, though, much sensitive and valuable information is moved over the web, so sending it all unprotected over the network is clearly a bad idea. Rather than come up with a fresh implementation of security for HTTP, however, HTTPS takes the existing HTTP definition and connects it to SSL/TLS. SSL takes care of establishing a secure connection, including authenticating the web server using the certificate approach discussed earlier and establishing a new symmetric encryption key known only to the client and server. Once the SSL connection is established, all subsequent interactions between the client and server use the secured connection. To a large extent, HTTPS is simply HTTP passed through an SSL connection.

That does not devalue the importance of HTTPS, however. In fact, it is a useful object lesson. Rather than spend years in development and face the possibility of the same kinds of security flaws that other developers of security protocols inevitably find, HTTPS makes direct use of a high quality transport security tool, thus replacing an insecure transport with a highly secure transport at very little development cost.

HTTPS obviously depends heavily on authentication, since we want to be sure we aren't communicating with malicious web sites. HTTPS uses certificates for that purpose. Since HTTPS is intended primarily for use in web browsers, the certificates in question are gathered and managed by the browser. Modern browsers come configured with the public keys of many certificate signing authorities (CAs, as we mentioned earlier). Certificates for web sites are checked against these signing authorities to determine if the certificate is real or bogus. Remember, however, what a certificate actually tells you, assuming it checks out: that at some moment in time the signing authority thoughts it was a good idea to vouch that a particular public key belongs to a particular party. There is no implication that the party is good or evil, that the matching private key is still secret, or even that the certificate signing authority itself is secure and uncompromised, either when it created the certificate or at the moment you check it. There have been real world problems with web certificates for all these reasons. Remember also that HTTPS only vouches for authenticity. An authenticated web site using HTTPS can still launch an attack on your client. An authenticated attack, true, but that won't be much consolation if it succeeds.

Not all web browsers always supported HTTPS, typically because they didn't have SSL installed or configured. In those cases, a web site using HTTPS only would not be able to interact with the client, since the client couldn't set up its end of the SSL socket. The standard solution for web servers was to fall back on HTTP when a client claimed it was unable to use HTTPS. When the server did so, no security would be applied, just as if the server wasn't running HTTPS at all. As ability to support HTTPS in browsers and client machines has become more common, there has been

a push towards servers insisting on HTTPS, and refusing to talk to clients who can't or won't speak HTTPS. This approach is called HSTS (HTTP Strict Transport Security). HSTS is an option for a web site. If the web site decides it will support HSTS, all interactions with it will be cryptographically secured for any client. Clients who can't or won't accept HTTPS will not be allowed to interact with such a web site. HSTS is used by a number of major web sites, including Google's `google.com` domain, but is far from ubiquitous as of 2020.

While HTTPS is primarily intended to help secure web browsing, it is sometimes used to secure other kinds of communications. Some developers have leveraged HTTP for purposes rather different than standard web browsing, and, for them, using HTTPS to secure their communications is both natural and cheap. However, you can only use HTTPS to secure your system if you commit to using HTTP as your application protocol, and HTTP was intended primarily to support a human-based activity. HTTP messages, for example, are typically encoded in ASCII and include substantial headers designed to support web browsing needs. You may be able to achieve far greater efficiency of your application by using SSL, rather than HTTPS. Or you can use SSH.

SSH

SSH stands for **Secure Shell** which accurately describes the original purpose of the program. SSH is available on Linux and other Unix systems, and to some extent on Windows systems. SSH was envisioned as a secure remote shell, but it has been developed into a more general tool for allowing secure interactions between computers. Most commonly this shell is used for command line interfaces, but SSH can support many other forms of secure remote interactions. For example, it can be used to protect remote X Windows sessions. Generally, TCP ports can be forwarded through SSH, providing a powerful method to protect interactions between remote systems.

SSH addresses many of the same problems seen by SSL, often in similar ways. Remote users must be authenticated, shared encryption keys must be established, integrity must be checked, and so on. SSH typically relies on public key cryptography and certificates to authenticate remote servers. Clients frequently do not have their own certificates and private keys, in which case providing a user ID and password is permitted. SSH supports other options for authentication not based on certificates or passwords, such as the use of authentication servers (such as Kerberos). Various ciphers (both for authentication and for symmetric encryption) are supported, and some form of negotiation is required between the client and the server to choose a suitable set.

A typical use of SSH provides a good example of a common general kind of network security vulnerability called a **man-in-the-middle** attack. This kind of attack occurs when two parties think they are communicating directly, but actually are communicating through a malicious third

party without knowing it. That third party sees all of the messages passed between them, and can alter such messages or inject new messages without their knowledge[7].

Well-designed network security tools are immune to man-in-the-middle attacks of many types, but even a good tool like SSH can sometimes be subject to them. If you use SSH much, you might have encountered an example yourself. When you first use SSH to log into a remote machine you've never logged into before, you probably don't have the public key associated with that remote machine. How do you get it? Often, not through a certificate or any other secure means, but simply by asking the remote site to send it to you. Then you have its public key and away you go, securely authenticating that machine and setting up encrypted communications. But what if there's a man in the middle when you first attempt to log into the remote machine? In that case, when the remote machine sends you its public key, the man in the middle can discard the message containing the correct public key and substitute one containing his own public key. Now you think you have the public key for the remote server, but you actually have the public key of the man in the middle. That means the man in the middle can pose as the remote server and you'll never be the wiser. The folks who designed SSH were well aware of this problem, and if you ever do use SSH this way, up will pop a message warning you of the danger and asking if you want to go ahead despite the risk. Folk wisdom suggests that everyone always says "yes, go ahead" when they get this message, including network security professionals. For that matter, folk wisdom suggests that all messages warning a user of the possibility of insecure actions are always ignored, which should suggest to you just how much security benefit will arise from adding such confirmation messages to your system.

SSH is not built on SSL, but is a separate implementation. As a result, the two approaches each have their own bugs, features, and uses. A security flaw found in SSH will not necessarily have any impact on SSL, and vice versa.

57.8 Summary

Distributed systems are critical to modern computing, but are difficult to secure. The cornerstone of providing distributed system security tends to be ensuring that the insecure network connecting system components does not introduce new security problems. Messages sent between the components are encrypted and authenticated, protecting their privacy and integrity, and offering exclusive access to the distributed service to the intended users. Standard tools like SSL/TLS and public keys

[7] Think back to our aside on Diffie-Hellman key exchange and the fly in the ointment. That's a perfect case for a man-in-the-middle attack, since an attacker can perhaps exchange a key with one correct party, rather than the two correct parties exchanging a key, without being detected.

distributed through X.509 certificates are used to provide these security services. Passwords are often used to authenticate remote human users.

Symmetric cryptography is used for transport of most data, since it is cheaper than asymmetric cryptography. Often, symmetric keys are not shared by system participants before the communication starts, so the first step in the protocol is typically exchanging a symmetric key. As discussed in previous chapters, key secrecy is critical in proper use of cryptography, so care is required in the key distribution process. Diffie-Hellman key exchange is commonly used, but it still requires authentication to ensure that only the intended participants know the key.

As mentioned in earlier chapters, building your own cryptographic solutions is challenging and often leads to security failures. A variety of tools, including SSL/TLS, SSH, and HTTPS, have already tackled many of the challenging problems and made good progress in overcoming them. These tools can be used to build other systems, avoiding many of the pitfalls of building cryptography from scratch. However, proper use of even the best security tools depends on an understanding of the tool's purpose and limitations, so developing deeper knowledge of the way such tools can be integrated into one's system is vital to using them to their best advantage.

Remember that these tools only make limited security guarantees. They do not provide the same assurance that an operating system gets when it performs actions locally on hardware under its direct control. Thus, even when using good authentication and encryption tools properly, a system designer is well advised to think carefully about the implications of performing actions requested by a remote site, or providing sensitive information to that site. What happens beyond the boundary of the machine the OS controls is always uncertain and thus risky.

References

[H14] "The Heartbleed Bug" by `http://heartbleed.com/`. *A web page providing a wealth of detail on this particular vulnerability in the OpenSSL implementation of the SSL/TLS protocol.*

[I12] "Information technology – Open Systems Interconnection – The Directory: Public-key and Attribute Certificate Frameworks" ITU-T, 2012. *The ITU-T document describing the format and use of an X.509 certificate. Not recommended for light bedtime reading, but here's where it's all defined.*

[NT94] "Kerberos: An authentication service for computer networks" by B. Clifford Neuman and Theodore Ts'o. IEEE Communications Magazine, Volume 32, No. 9, 1994. *An early paper on Kerberos by its main developers. There have been new versions of the system and many enhancements and bug fixes, but this paper is still a good discussion of the intricacies of the system.*

[P16] "The International PGP Home Page" `http://www.pgpi.org`, 2016. *A page that links to lots of useful stuff related to PGP, including downloads of free versions of the software, documentation, and discussion of issues related to it.*