**File Security System for Operating Systems**

**Introduction**

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

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

This point is especially important for operating systems. You might not care about the security of a particular web server or database system if you don’t run that software, and you might not care about the security of some middleware platform that you don’t use, but everyone runs an operating system, 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. 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. So large, complex programs are likely to be harder to secure than small, simple programs. Not many other programs are as large and complex as a modern operating system.

Another challenge in securing operating systems is that they are, for the most part, meant to support multiple processes simultaneously. As you’ve learned, there are many mechanisms in an operating system meant to segregate processes from each other, and to protect shared pieces of hardware from being used in ways that interfere with other processes. If 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 unwritable does not get altered. If the applications cannot rely on proper implementation of OS abstraction security guarantees, then they cannot use these abstractions to achieve their own security goals. At the minimum, that implies a great deal more work on the part of the application developers, since they will need to take extra measures to achieve their desired security goals. Taking into account our earlier discussion, they will often be unable to achieve these goals if the abstractions they must rely on (such as virtual memory or a welldefined 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.

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

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

**What Are We Protecting?**

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

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

• examine or alter any process’ memory

• read, write, delete or corrupt any file on any writeable persistent storage medium, including hard disks and flash drives

• change the scheduling or even halt execution of any process

• send any message to anywhere, including altered versions of those a process wished to send

• enable or disable any peripheral device

• give any process access to any other process’ resources

• arbitrarily take away any resource a process controls

• respond to any system call with a maximally harmful lie

In essence, processes are at the mercy of the operating system. It is nearly impossible for a process to “protect” any part of itself from a malicious operating system. We typically assume our operating system is not actually malicious1 , 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.

**Security Goals and Policies**

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

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

• Confidentiality – Keep your secrets. If some piece of information is supposed to be hidden from others, don’t allow them to find it out. For example, you don’t want someone to learn what your credit card number is – you want that number kept confidential.

• Integrity – If some piece of information or component of a system is supposed to be in a particular state, don’t allow an 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’ memory space cannot be arbitrarily read by another process. We might have an integrity goal stating that if a user writes a record to a particular file, another user who should not be able to write that file can’t change the record. We might have an availability goal stating that one process running on the system cannot hog the CPU and prevent other processes from getting their share of the CPU. If you think back on what you’ve learned about the process abstraction, memory management, scheduling, file systems, IPC, and other topics from this class, you should be able to think of some other obvious confidentiality, integrity, and availability goals we are likely to want in our operating systems.

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

**ASIDE: SECURITY VS. FAULT TOLERANCE**

When discussing the process abstraction, we talked about how virtualization protected a process from actions of other processes. For instance, we did not want our process’ memory to be accidentally overwritten by another process, so our virtualization mechanisms had to prevent such behavior. Then we were talking primarily about flaws or mistakes in processes. Is this actually any different than worrying about malicious behavior, which is more commonly the context in which we discuss security? Have we already solved all our problems by virtualizing our resources? Yes and no. (Isn’t that a helpful phrase?) Yes, if we perfectly virtualized everything and allowed no interactions between anything, we very likely would have solved most problems of malice. However, most virtualization mechanisms are not totally bulletproof. They work well when no one tries to subvert them, but may not be perfect against all possible forms of misbehavior. Second, and perhaps more important, we don’t 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 . . .

The Basics of OS Security In a typical operating system, then, we have some set of security goals, centered around various aspects of confidentiality, integrity, and availability. Some of these goals tend to be built in to the operating system model, while others are controlled by the owners or users of the system. The built-in goals are those that are extremely common, or must be ensured to make the more specific goals achievable. Most of these built-in goals relate to controlling process access to pieces of the hardware. That’s because the hardware is shared by all the processes on a system, and unless the sharing is carefully controlled, one process can interfere with the security goals of another process. Other built-in goals relate to services that the operating system offers, such as file systems, memory management, and interprocess communications. If these services are not carefully controlled, processes can subvert the system’s security goals. Clearly, a lot of system security is going to be related to process handling. If the operating system can maintain a clean separation of processes that can only be broken with the operating system’s help, then neither shared hardware nor operating system services can be used to subvert our security goals. That requirement implies that the operating system needs to be careful about allowing use of hardware and of its services. In many cases, the operating system has good opportunities to apply such caution. For example, the operating system controls virtual memory, which in turn completely controls which main memory addresses each process can access. Hardware support prevents a process from even naming a physical memory address that is not mapped into its virtual memory space. (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.) TIP: THE WEAKEST LINK It’s worthwhile to remember that the people attacking your systems share many characteristics with you. In particular, they’re probably pretty smart and they probably are kind of lazy, in the positive sense that they don’t do work that they don’t need to do. That implies that attackers tend to go for the easiest possible way to overcome your system’s security. They’re not going to search for a zero-day buffer overflow if you’ve chosen “password” as your password to access the system. The practical implication for you is that you should spend most of the time you devote to securing your system to identifying and strengthening your weakest link. Your weakest link is the least protected part of your system, the one that’s easiest to attack, the one you can’t hide away or augment with some external security system. Often, a running system’s weakest link is actually its human users, not its software. You will have a hard time changing the behavior of people, but you can design the software bearing in mind that attackers may try to fool the legitimate users into misusing it. Remember that principle of least privilege? If an attacker can fool a user who has complete privileges into misusing the system, it will be a lot worse than fooling a user who can only damage his own assets. 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 [SC00]. System calls offer the operating system another opportunity to provide protection. In most operating systems, processes access system services by making an explicit system call, as was discussed in earlier chapters. As you have learned, system calls switch the execution mode from the processor’s user mode to its supervisor mode, invoking an appropriate piece of operating system code as they do so. That code can determine which process made the system call and what service the process requested. Earlier, we only talked about how this could allow the operating system to call the proper piece of system code to perform the service, and to keep track of who to return control to when the service had been completed. But the same mechanism gives the operating system the opportunity to check if the requested service should be allowed under the system’s security policy. Since access to peripheral devices is through device drivers, which are usually also accessed via system call, the same mechanism can ensure proper application of security policies for hardware access. When a process performs a system call, then, the operating system will use the process identifier in the process control block or similar structure to determine the identity of the process. The OS can then use access control mechanisms to decide if the identified process is authorized to perform the requested action. If so, the OS either performs the action itself on behalf of the process or arranges for the process to perform it without further system intervention. If the process is not authorized, the OS can simply generate an error code for the system call and return control to the process, if the scheduling algorithm permits