CS 161 Computer Security
Fall 2005 Joseph/Tygar/Vazirani/Wagner **Notes 30**

1 Isolation

The topic for today is *isolation*. A program is isolated if it cannot affect other programs on the system. Thus, isolation refers to an inability to causally influence other programs on the system.

Isolation is related to some topics we have seen before, such as access control. One difference is that access control is a mechanism for enforcing some security policy (a means to an end), whereas isolation is a security goal (the end itself). Isolation is also related to previous lectures on virtual memory and memory protection. The difference is that virtual memory seeks to provide only memory isolation between processes (each process receives a disjoint address spaces and cannot affect other processes through memory reads and writes). Memory protection does not prevent other kinds of influence, such as opening an IPC connection from one process to another. When we want to isolate a process, we want to isolate against *all* influences, so memory protection alone is not enough.

What are the applications of isolation? Here are a few.

- I run across a cool piece of software that will draw dancing pigs on the screen, and I want to download it and try it, but I don't know whether I can trust the developer. It would be great to be able to run it in an isolated environment, where it cannot harm the rest of my machine even if it contains malicious code or bugs.

  This is often known as the *sandboxing* problem. We want to give the downloaded software its own little sandbox where it can do whatever it wants, as long as it doesn't escape the sandbox. If it tries to do anything disruptive, the effect will be limited to its sandbox, so the worst that can happen is it will disrupt itself.

- I want to display a MS Word file that someone emailed me, but I don't want it to be able to infect my machine with a macro virus. It would be great if I could run an sandboxed instance of MS Word to view just this document, with no fear that it will trash my other Word documents.

- I'm designing a complicated software application. Following the principle of least privilege, I want to decompose the application into multiple pieces. Each piece should be isolated from the others, so that if one piece is penetrated, the integrity of the others will be preserved.

Soon I will start to show you a number of different ways to try to enforce a policy of isolation, but first let's explore the software decomposition issue a bit more to understand some of the requirements.

2 Decomposing Software for Security

I've talked before about the relevance of modularity to application security, but let me now show you how to select a decomposition of your system into modules that will be helpful for security.

I'll work from a case study: building a secure mailer. At one point, one of the most popular mail daemons was sendmail, an application written by Eric Allman while he was a staff member here in Berkeley EECS. Sendmail is a large, monolithic application, consisting over 100K lines of C code. It runs as root. Unfortunately, it has been plagued by security problems—and because it runs with root privilege, each one of those security holes exposes the entire machine to mischief by the intruder.

qmail is a secure mailer written by Dan Bernstein, partly in response to these problems. qmail is now the second most popular mail daemon on the Internet. Interestingly, in 1997 Bernstein offered a $500 prize to the first person to find a serious security hole in his system. The $500 still remains unclaimed. Let's see why qmail has fared so much better than sendmail.

What does a SMTP mail daemon need to do?

- It receives incoming email via SMTP connections to port 25 on this machine. Therefore, the mail daemon has to listen for connections to port 25 on this machine and be prepared to receive email from other hosts.
- It receives email submissions from other programs running on this host. Therefore, it has to be prepared to be invoked by other programs who want to submit mail for transmission elsewhere.
- When it receives an email message, it has to queue the message, determine where to route this message (it may require delivery to a user on this host, or it may be forwarded the SMTP daemon on some other host), and deliver the email message to a local user if necessary.

Internally, qmail is broken down into separate programs that each perform a simple task. Here are the programs that make up qmail (see Figure 1):

- qmail-smtpd: This program listens for incoming network connections on port 25. It speaks the SMTP protocol, gathers up the email message submitted to this host, calls qmail-queue to add the message to the queue, and writes a log file. qmail-smtpd has its own user account (qmaild), and runs in the background with the permissions of that user account.
- qmail-inject: This program is executed by local programs that want to submit email messages for transmission. It accepts a message on stdin and then calls qmail-queue to add the message to the queue. qmail-inject runs with the permissions of whatever user invokes it.
- qmail-queue: This program reads a message from its input and appends it to the mail queue. The queue directory is owned by a special user account (qmailq), and qmail-queue is a setuid program, so that when it is executed it takes on the permissions of that special account. qmail-queue is the only program that can write the queue directory on the file system, and it has the responsibility of ensuring that the queue data structure on disk is always consistent and cannot be corrupted.
- qmail-send: This program runs in background, waiting for the queue to become non-empty. Whenever the queue contains a message, qmail-send reads a message from the queue, determines whether it should be routed to a local user or to a remote host, and executes either qmail-lspawn or qmail-rspawn accordingly. qmail-send also runs with the same permissions as qmail-queue (namely, as user qmailq).
- qmail-lspawn: This program accepts an email message on its input, looks up the user ID who this message should be delivered to, becomes that user (giving up all other permissions it once had, and now executing with the permissions of that user ID), and then executes qmail-local. qmail-lspawn is a setuid-root program, so that when it is executed it initially takes on root permissions (which are needed so that it can become the appropriate user).

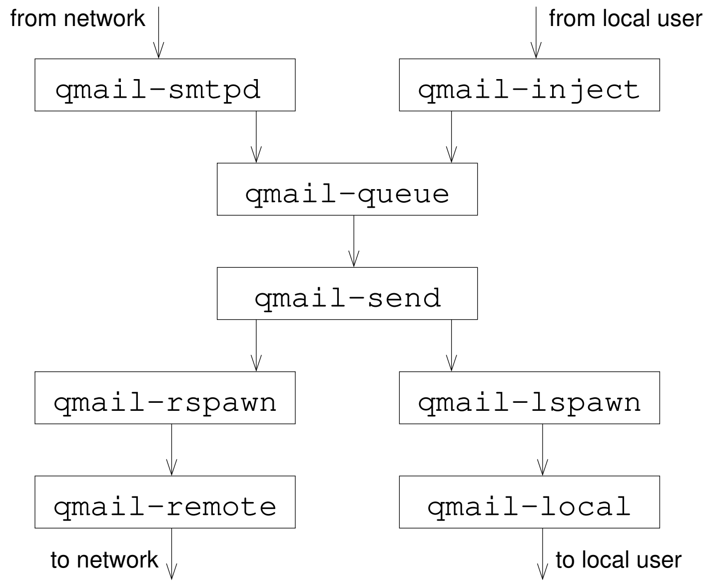

Figure 1: An overview of the qmail architecture.

- qmail-local: This program actually delivers the email message by appending it to the user's mailbox file (or by passing it off to a filter, if so specified by the user's `~/.forward` file). qmail-local runs with the permissions of the user who is receiving the email.
- qmail-rspawn: This program accepts an email message as input, determines the remote host who this message should be transmitted to, and invokes qmail-remote to actually transmit it to that host. It is setuid to its own special user ID (qmailr).
- qmail-remote: This program accepts an email message as input, opens a network connection to port 25 on that remote host, and speaks the SMTP protocol to deliver the message to the remote SMTP daemon. It runs with the permissions of the invoking user (qmailr).
- qmail-start: This program starts up all the long-running processes, with the appropriate user IDs.

Why does qmail use so many programs?

- This lets qmail minimize the amount of code running as root. All of sendmail runs as root. In contrast, in qmail, only qmail-start and qmail-lspawn run as root. Security holes in other programs do not give the intruder root-level access to your machine; they're still worth avoiding, but the harm is reduced. This is an instance of the principle of least privilege.

  Notice how this relies on a certain level of isolation between the qmail programs. If the non-root programs could tamper with the memory of the root programs (for instance), this property would be worthless. Isolation lets us cordon off the root code from the rest of the application.
- It reduces the amount of code that is security-critical. For instance, qmail-inject can only be invoked by local users. If it has a buffer overrun vulnerability, that's unfortunate, but at least there is no direct way for a remote attacker to exploit this buffer overrun.
- Logically different functions are separated into mutually distrusting programs. No program trusts data from the others. Thus, even if some subset of the programs are broken, so that they can be completely controlled by an intruder, the intruder still can't attack the other programs or take over your system.

(Exception: If an intruder compromises qmail-lspawn, you're hosed, because qmail-lspawn runs as root.) For instance, even if the attacker manages to gain control of qmail-smtpd, qmail-inject, qmail-local, qmail-rspawn, and qmail-remote, it still cannot gain access to the queue files, because the queue files are only accessible to user qmailq, and only qmail-queue and qmail-send run with the permissions of that user.

Notice how this relies on isolation between the programs. If one program could take control of another qmail program (say, by issuing debugging commands, or by tampering with its binary executable on disk, or by any of numerous other methods), then we would be unable to provide this level of protection.

- Each program is extremely simple. The largest is qmail-send, which consists of 1600 lines of C code. All of the others use less than 800 lines of code.

In short, qmail uses many little programs because this is the easiest way to get isolation in Unix. Each program is (at least partially) isolated from all the others. And isolation is what lets us meaningfully decompose the application into "watertight compartments" that minimize the spread of damage. In effect, it provides a firewall between each module of the application.

These examples also make clear a more fundamental point about sandboxing and isolation: pure isolation is usually too strict. Much more commonly, we need to combine *isolation* with *controlled sharing*. Isolation is a starting point that is analogous to the deny-all starting point of a default-deny policy. Controlled sharing is about allowing limited escape routes out of the sandbox, so that one can interact usefully with the sandboxed application (hopefully without exposing ourselves to attack). In the case of qmail, the controlled sharing between the qmail programs occurs primarily through explicit communication channels between pairs of programs.

Let's try another example to help see how to decompose applications for security. We'll design a software architecture for a simple web service that converts files from one format to another—say, from MP3 to OGG. This web service needs to accept incoming connections to port 80, receive the contents of a file, perform some complicated computations to translate the file into the new format, and then send the result back out over the network. Thus, we might break this down into two pieces: the master, a process that receives files on port 80, invokes the slave with the contents of this file, and sends the slave's output over the network; and the slave, which takes as input a byte array, interprets its input as an MP3, transforms the MP3 into OGG format, and produces as output another byte array containing the OGG data. Notice that the slave is computing a deterministic function of its input, and does not need any permissions at all (not to read files, or to access the network, or to do anything else at all except compute). Thus, we can arrange that all the complicated code is sandboxed in a process that cannot do anything at all but pass bytes back and forth to the master. Consequently, if that complicated MP3-to-OGG code is buggy, the worst that can happen is that people will receive incorrect or bogus OGG files, but no harm can happen to our machine.

Here's another example for you to ponder. You're writing a web browser. You want to decompress a file that was received from across the network, but the decompression program looks fairly complex and you're not 100% sure whether you can trust it. How do you structure your application to minimize trust in the decompressor? (This is an example with some historical merit. In 2002, it was discovered that any program that used versions of the zlib decompression library dating from Feb 1998–Mar 2002 were vulnerable to a code injection exploit.)

Today, this kind of software design is not especially common, perhaps because common operating systems (including Unix and Windows) don't make it very easy to get the kind of isolation needed. Nonetheless, as we've seen, there are many applications where sandboxing and isolation would be very useful. The goal of a secure sandbox is that it should be inescapable: even if the program running in the sandbox is malicious (or controlled by an intruder), that program should not be able to escape the sandbox. So, how can we build a secure sandbox? How can we enforce isolation guarantees? The rest of this lecture will discuss several techniques for doing exactly this.

3 Access Control

The most obvious way to build a sandbox is to create a new user account, install the sandboxed program in this account, log in as that user, and run the program. This effectively uses the operating system's access control mechanism to control what the program can do. As a generalization, we could use any other way at our disposal to limit the permissions available to the sandboxed program to the minimum needed to set up an isolated sandbox.

The problem with this approach is that, in most of today's operating systems, the access control mechanism is primarily focused on protecting files. For instance, in Unix any program can create a new network connection and can run a server on any local port greater than 1024. Moreover, there is no way to remove this privilege. In other words, not only does Unix have a "default allow" policy when it comes to creating network connections, but there is not even any way to deny a process these privileges. Another difficulty is that, on a Unix machine, many files are world-readable, which corresponds to a "default allow" policy, and we would need to arrange that sandboxed programs do not needlessly receive access to those files.

This makes it hard to build a secure sandbox. For instance, a malicious program running in the sandbox could create a network connection to attack other machines, to spread a worm, or to send spam that looks like it came from my machine. If my host is located behind a firewall, the malicious program can scan and attack other machines behind the same firewall, which is something that a remote attacker would not be able to do on his own. It might be able to steal a copy of the /etc/passwd file on your machine and email it through an anonymous remailer back to the intruder, thereby revealing the names of all users on the system.

Notice that qmail essentially uses this strategy to build its sandbox. Therefore, its isolation guarantees are actually slightly weaker than mentioned before. If an intruder manages to gain control of one of the qmail programs, the intruder is not entirely isolated—there are some unfortunate things the intruder can do (such as attacking other hosts on the same intranet). This is a limitation of the isolation strategy that qmail uses; unfortunately, it would be difficult for qmail to do much better while remaining portable.

4 System Call Interposition

One solution is to use *interposition* on the system call interface. In other words, we place a sandbox enforcer between the sandboxed application and the operating system, and have the enforcer mediate all system call requests. When the sandboxed application tries to issue a system call, that syscall is actually re-directed to the enforcer, and the enforcer is given the chance to approve or deny the syscall request according to the arguments to the syscall. This provides a way to extend the operating system's access control policy without modifying the OS itself.

Notice the crucial property of system calls is that, if we set up things appropriately, the sandboxed application cannot affect anyone else without calling a system call. Consequently, interposing on the system call interface is sufficient to ensure complete mediation.

How does the enforcer decide which system calls to allow? That depends on our sandboxing policy. Here are some example policies.

- Consider the MP3-to-OGG example given earlier, where we wish to allow the sandboxed slave to perform pure computation and nothing more. In this case, we can deny almost all system calls. To allow the sandboxed slave to receive input and produce output, we start it up with file descriptors 0 and 1 (representing stdin and stdout) connected to the master, and we allow the slave to perform `read()` and `write()` system calls—but nothing else. Note that since the slave cannot call `open()` or `connect()`, it cannot open files or network connections.

- Suppose we want to run the Adobe Acrobat PDF viewer in a sandbox, so that any security holes in it cannot harm the rest of the system. We might allow the sandboxed Acrobat to `connect()` to port 6000 on localhost (so that it can open a window on our X Windows display) and to `open()` or otherwise manipulate any file under `~/.acrobat` (so that it can maintain its preferences). We might also allow it to freely call `read()` or `write()` (since these will only do anything useful on an open file descriptor, and the rest of the policy limits which file descriptors can be opened). This is appealingly simple—but then reality intervenes. Acrobat loads dynamic libraries, so we need to allow it to `open()` various libraries for reading and `mmap()` them into memory. It opens `/usr/lib/locale` to determine what language to use, so we have to allow opening that file for reading as well. It uses signals, so we have to allow various signal-related syscalls. It uses threads, so we have to let it spawn new processes, and we have to make sure to apply system call interposition to those processes as well. The sandboxing policy needed is surprisingly complex. But, interestingly, it is possible to successfully sandbox many common programs in this way, after some configuration of the policy.

This might sound fairly simple, but there are some subtle pitfalls here. It is very easy to end up with TOCTTOU vulnerabilities, where the meaning of the system call when it is inspected by the enforcer is different from its meaning when it is actually executed by the OS kernel. Here are two examples:

- The `open()` system call's first argument is a pointer to a filename. If we pass this pointer to the enforcer, and then have the enforcer read the filename from the sandboxed program's memory, then a malicious program could arrange for this filename to change after it has been validated by the enforcer but before the OS executes the `open()` call. One solution to this is to have the OS copy the filename into kernel memory, and then copy that filename to the enforcer.

- When the program calls `open("foo")`, the current working directory affects precisely which file gets opened. If the sandboxed program is multi-threaded or running on a multi-processor (SMP) machine, it might be able to change its working directory between the time when the enforcer validates the `open()` and when it is executed. For instance, suppose the sandboxed program starts within the `~/.acrobat` directory, calls `open("key", O_RDONLY)`, and then changes its working directory to `~/.ssh2` after the `open()` call is approved and before it is executed. If the sandboxed program wins the race condition, it gains the ability to read `~/.ssh2/key`—even if that was prohibited by the sandboxing policy.

The latter example is representative of a more general problem with *shadow state*. The OS maintains some state for each running process, such as its current working directory. To make security decisions, the enforcer has to independently maintain its own copy of this state (e.g., by observing all `chdir()` system calls). If the two copies of state get out of sync, then the enforcer may allow some system calls that were intended to be prohibited by the sandboxing policy. This is a general problem any time you try to interpose a reference monitor on an interface where the meaning of a call depend on some state or context not made manifest in the arguments of the call.

One approach to dealing with this is to have the enforcer *virtualize* the operating system interface. Rather than merely passively observing system calls and approving or denying them, the enforcer might receive system calls, emulate what an operating system might do upon receiving that system call, and respond to the sandboxed program accordingly. The emulation process might require the enforcer to make its own system calls to the underlying operating system. Thus, the sandboxed application thinks it is running on top of a real operating system (when actually it is only running on an emulated OS, as implemented by the enforcer); to the sandboxed app, the enforcer implements the same interface that an OS would, while to the real OS the enforcer just looks like any other process. One consequence is that all of the state related to the sandboxed process would be maintained by the enforcer. For instance, when the sandboxed process issues a `chdir()` system call, the enforcer updates an entry listing the working directory for the sandboxed process, and then returns "Success" to the sandboxed process without passing the `chdir()` syscall on to the real OS. When the sandboxed process issues a `open("foo")` syscall, the enforcer might translate this into `open("`*cwd*`/foo")` where *cwd* is taken from the enforcer's records of the sandboxed process's current working directory.

There has been detailed research into system call interposition. There are many interesting details that I've omitted, but this gives you some idea of how these techniques tend to work. If you're interested in more details, you can read about tools such as Systrace, Janus, and Ostia.

Question for thought: How could you use system call interposition to make qmail more bullet-proof?

5 Physical Isolation

Another simple strategy for building a secure sandbox is to run the sandboxed program on a physically isolated machine. For instance: We drive down to Fry's, buy a cheap new machine, and install and run the sandboxed program on this standalone machine. When we're done, we can reboot and reformat the dedicated machine and reuse it to run another sandboxed program.

In general, physical isolation is often a good way to achieve isolation. If I take a dedicated machine and physically separate it from everything else, I can be pretty confident that nothing can escape the sandbox. If the dedicated machine doesn't have any wires connecting it to anything else, it probably can't communicate. (Ok, I need to be wary of wireless networking, so perhaps I should buy a laptop, disconnect the power cord so it runs off its battery, and place it in a Faraday cage. You get the idea.) Marcus Ranum once wrote that ultimate high-assurance firewall could be installed using a pair of wirecutters, and he was only half-joking.

The obvious problem with physical isolation is that it can be expensive: you have to buy a dedicated machine for every program you want to sandbox. Still, it can be useful in some settings. For instance, the military has long used a strategy like this to provide intelligence analysts with access both to the outside Internet (where classified material is forbidden) and to the military's internal SIPRNET (a private network that carries classified data). Each analyst receives two machines sitting on their desk: one is connected to the Internet, one to the SIPRNET, and they are not connected to each other.

Even if it is not very practical, it turns out that physical isolation is a very useful mental metaphor for thinking about the design of secure systems. When it comes time to decompose an application into multiple pieces, I often imagine executing each module on its own machine, and then connecting up each pair of modules that must communicate by a point-to-point wire between the corresponding computers. This makes explicit all causal connections between the modules, so I don't forget any interactions.

6 Virtual Machines

If real machines are too expensive, one solution is to use a *virtual machine*. A virtual machine is a software application that emulates the behavior of a physically separate machine. Some well-known examples of virtual machines include VMWare, Virtual PC, QEmu, and Bochs.

How does a virtual machine work? The full details are too complicated for this course, but here is a brief introduction. Imagine building an x86 emulator: a program that takes as input an x86 binary (which is just a sequence of x86 instructions) and interprets the effect those would have *entirely in software*. Thus the emulator maintains (in software) the emulated state of an x86 CPU. The only tricky question is what to do when the emulated program tries to do I/O: say, to read a sector from the hard disk. What a virtual machine does is to also emulate the behavior of physical devices as well. For instance, if I install VMWare on Linux and use VMWare to emulate some x86 program, then VMWare might set aside a 100MB file on my Linux filesystem to represent the state of an emulated hard drive with 100MB of capacity. When the x86 program tries to read or write a sector to disk, VMWare translates that into a read or write to the 100MB file, issuing an appropriate syscall to Linux to perform that operation. When the x86 program tries to write to its screen, VMWare might translate that into writing some pixels on a window on my Linux desktop. As described so far, this emulation strategy would incur a tremendous performance slowdown, but sophisticated techniques have been developed for running emulated programs at almost full speed.

Virtual machines give us a way that a single physical computer can simulate the presence of dozens of computer. Consequently, virtual machines give us all the benefits of physical isolation, but without having to buy lots of hardware from Fry's. For instance, if I want to run mobile code of unknown provenance, I might start up a virtual machine and execute the mobile code within the virtual machine, confident that the mobile code cannot cause any lasting harm to my physical desktop. For instance, I could delete all state associated with the virtual machine after running the mobile code, preventing it from having any lasting effect on my files and my machine.

Virtualization is a powerful technique. We have seen how virtual memory virtualizes the memory abstraction (providing each process with the illusion that it has its own RAM and its own address space); how system call interposition can virtualize the operating system syscall interface; and now how virtual machines can virtualize the hardware interface.

7 Interpreted Code

As the virtual machine example might illustrate, interpreted languages can be used for sandboxing. An interpreter is just a big loop that repeatedly decodes and executes a sequence of instructions in some language.

Perhaps the simplest example of this comes from combinatorial circuits. Recall that any stateless deterministic computation can be implemented as a combinatorial circuit. To put it another way, given any boolean function $f : \{0,1\}^n \to \{0,1\}$, we can find a combinatorial circuit that computes f. In this context, a combinatorial circuit is a network of gates (e.g., AND, OR, NOT) that has n inputs, 1 output, and no cycles or memory. When a computation is expressed as a combinatorial circuit, it is very easy to sandbox this computation: we can simply evaluate the circuit in software. We know that this computation cannot perform any I/O, gain access to secrets, or affect the rest of our machine, simply because it is computing a pure, deterministic, side-effect-free function f. Such functions cannot perform any input (it is impossible for them to depend on anything other than the n inputs explicitly listed) and cannot perform any output (they have no side effects), so they are totally sandboxed.

Let's take an example. Suppose we want to build an extensible spam-filtering application. Maybe Sam has studied the characteristics of spam and built a program that takes an email as input and classifies it either as "spam" or "not spam". I'd like to download and use his program to help filter my email, but I don't want to run a program from an untrusted source directly in my account, because for all I know it might be a malicious Trojan horse. Note that Sam's program can be expressed as a boolean function f that takes an email (which is just a bit-string) as input and produces a boolean output. Thus, one solution would be to have Sam express his program as a combinatorial circuit and make the circuit available on his web page. I could download his circuit and evaluate it on each incoming email. Even if Sam is malicious, his filter cannot leak the contents of any of my emails to Sam or anyone else (except possibly for whether Sam's filter classified each email as spam or not), and it cannot harm my machine—the worst it can do is cause me to make the wrong filtering decisions.

An interpreter for boolean circuits can be made extremely simple. As we know, NAND gates are all you need to express arbitrary combinatorial logic. We can thus express such a circuit as a sequence of instructions for a very simple CPU, as follows. We store each value in the circuit in its register. Each instruction reads inputs from two specified registers, computes their NAND, and stores it to a third register (e.g., `NAND r1037, r27, r45` might compute the NAND of the bit in register r27 and the bit in r45, storing the result in register r1037). An interpreter for this language can be implemented in a few lines of code.

Of course, circuits aren't a very user-friendly or flexible programming language. However, we can apply the same general principles to other interpreted languages. The key trick is to design the language so that it is impossible to express operations that would violate the sandboxing policy. For instance, we could design a language so that there is no way to perform I/O or to read/write outside the program's address space, and gain the same benefits of pure, deterministic, side-effect-free computation as a sandboxing tool.

One good example of this is BPF, an interpreted language for expressing packet filters that can be downloaded into the kernel. The BPF language is designed so that you cannot even express harmful programs in the language. For instance, the designers wanted to make sure that users could not render the machine unusable by downloading a program with an infinite loop into the kernel, so the BPF language is designed to make it impossible to write programs with non-terminating loops by the simple expedient of forbidding all backward jumps (all of BPF's control-transfer instructions only permit forward jumps).