ANALYZING COMPUTER SECURITY

A THREAT / VULNERABILITY / COUNTERMEASURE APPROACH

CHARLES P. PFLEEGER | SHARI LAWRENCE PFLEEGER

ANALYZING COMPUTER SECURITY

Is Security Provision Easier in the Cloud? **Sidebar A-1**

Each of us knows how annoying and even expensive it can be to ensure that our devices and data are secure. From downloading updates to configuring firewalls, from changing passwords regularly to doing frequent scans, we devote time to security that we would rather spend on our primary jobs. And when a security incident occurs, we must report the problems, work with support staff to find the causes, lose time while the cures or countermeasures are put in place (including restoring data and systems), and slowly get back to what we were doing before the incident took place.

So it is very appealing to delegate many of the security activities to the cloud provider. In a SaaS cloud, a customer may download an application only for the time she needs it. The cloud provider makes sure that it is the most up-to-date version, with appropriate security releases and virus scanners. Similarly, in most clouds the provider configures the firewalls, applies the patches, and updates the antivirus signatures and software.

But from the provider's point of view, security may not be so easy. Think about the difficulty of providing security over the almost infinite variations in software and infrastructure offered by cloud providers. The cloud may shift the burden for security, but it doesn't always solve the problems.

that encircles the world? Similarly, how is data confidentiality maintained when the data can "live" in shared resources, especially when the data can be moved around without the customers' knowledge or control? Sidebar A-2 gives examples of these problems in the context of insider threat.

Technical Risks

Technical risks abound in the cloud. For example, there is the risk of resource depletion (sometimes called resource exhaustion): a customer requests a resource allocation, but nothing is available. How can this be a security risk, too? What about the need to isolate data or operations, when the resources are shared? And how can the cloud protect data in transit? Would encryption slow down the cloud's ability to be nimble, exchanging safety for sluggishness?

In Chapter 15, we discuss distributed denial-of-service attacks, where a huge volume of requests slows down service provision or even prevents access. How can such an attack be perpetrated on the cloud, and how could the provider institute appropriate countermeasures?

Finally, consider the system controlling the cloud itself. Like a giant operating system, it must control infrastructure provision and service provision, map scattered data together and identify their owners, and coordinate security and privacy policies and services. How could such a system be compromised from the inside? From the outside?

Are these technical risks real? Jansen and Grance [JAN11] point out several examples of cloud compromise. For instance, they report that, in 2009, a botnet was discovered operating from inside an IaaS cloud, and spammers have rented cloud space to launch phishing campaigns. And because clouds can marshal large numbers of resources, Jansen and Grance have warned that clouds can be put to work to break encryption quickly and have described how.

Insider Threat in the Cloud **Sidebar A-2**

In Chapter 5 we discuss the possibility that someone with authorized access to a system can harm data, users, and systems. Insiders can be found in the cloud, too. What kind of damage can they do?

One big insider risk is that customer instructions for deleting data may not be followed. Because the customer has no control over the cloud infrastructure, there may be no way to check that deleted data are really gone—they may just have been moved to an "undisclosed location." Similarly, a malicious insider can attack data integrity, changing actual values but manipulating the interface so that customers think the data have not been changed. What might that kind of attack do to a bank that stores its data in the cloud? Interest might be calculated on a smaller set of values, while the customer thinks his balance is larger.

So the cloud provides more places to hide, and the places can be reconfigured dynamically. Even if you know that an unwelcome incident has occurred, this dynamic reshuffling can make forensic analysis extremely difficult, even with an audit trail. And because the cloud can extend across many legal jurisdictions, something that is illegal in one part of the cloud may in fact be legal in another.

Legal Risks

In addition to some of the legal risks mentioned earlier, the cloud holds security risks related to identify management and licensing. How do we manage identities in the cloud? How can we authenticate an access, manage huge access control lists, or identify a user? The answers to these questions address important confidentiality and availability concerns.

Similarly, if licenses run out, support for applications could disappear. Suppose you are a customer of a cloud provider and your SaaS cloud enables you to develop the special-purpose application that is the source of your business's income. One day, your developers access the cloud to perform maintenance on the company product, only to discover that the software license has expired and maintenance is now impossible. Such expirations affect availability, one of the pillars of C-I-A.

6

My Cup Runneth Over

CHAPTER SPOTLIGHT

- Unchecked, excessive data transfer
- How operations on data can compromise software
- Defenses during program development
- Hardware memory protection
- Control of access to objects in general; the subject–object–mode paradigm

Buffer overflows are one of the most common security failures; they are the basis of four of the Mitre–SANS top 25 types of programming errors for 2010 [MIT10], even though the exploits of the vulnerability class have been around at least since the Morris worm of 1988 (which we describe later in this chapter).

The concept is simple: trying to put more than *n* bytes of data into a space big enough to hold only *n*. The countermeasure is equally simple: check the size of your data before trying to write it.

Unfortunately, programs write data frequently, sometimes without checking size; program function libraries contain code that fails to check and, in some situations, is impossible to check. Programmers assume no normal data would ever exceed a particular length, or someone took the program from one environment for another purpose, or nobody checked to see if a call for free memory (in which to write data) succeeded or failed, or a table grew without limit over time. Programs can run for a long time in unstressed conditions before this fault is manifested, and the result of the flaw can be difficult to trace back to its simple root cause.

Typically, a buffer overflow causes a program to fail catastrophically, because the extra data end up in a place that causes a serious problem with continued execution; the program terminates abnormally and the programmer or user has to try to discover and fix the problem. Worse is the situation in which the overflow effect depends on what else is in memory: The overflow does not have a consistent, repeatable effect, which makes deducing the cause nearly impossible. But occasionally, the overflow data just happen to fit smoothly into the flow of the program, and execution continues pretty much normally. This latter case is the security concern because the program continues, but with changed instructions or control parameters. As you will see, attackers welcome buffer overflows as a mechanism with which to insert instructions directly into the path of execution. That is, the attacker can choose any code and force the computer to execute those instructions.

We begin this chapter with a simple but real example of an attacker's search for a buffer overflow. (Actually, it was the probing of an honest computer security consultant.) Do not be fooled because the example is old; the method and outcome are just as valid today.

ATTACK: WHAT DID YOU SAY THAT NUMBER WAS?

In 1999, security analyst David Litchfield [LIT99] was intrigued by buffer overflows. He had both an uncanny sense for the kind of program that would contain overflows and the patience to search for them diligently. He happened onto the Microsoft Dialer program, dialer.exe.

Dialer was a program for dialing a telephone. Before cell phones, WiFi, broadband, and DSL, computers were equipped with modems by which they could connect to the land-based telephone network; a user would dial an Internet service provider and establish a connection across a standard voice telephone line. Many people shared one line between voice and computer (data) communication. You could look up a contact's phone number, reach for the telephone, dial the number, and converse; but the computer's modem could dial the same line, so you could feed the number to the modem from an electronic contacts list, let the modem dial your number, and pick up the receiver when your party answered. Thus, Microsoft provided Dialer, a simple utility program to dial a number with the modem. (As of 2010, dialer.exe was still part of Windows 7, although the buffer overflow described here was patched shortly after Litchfield reported it.)

Litchfield reasoned that Dialer had to accept phone numbers of different lengths, given country variations, outgoing access codes, and remote signals (such as to enter an extension number). But he also suspected there would be an upper limit. So he tried dialer.exe with a 20-digit phone number and everything worked fine. He tried 25 and 50, and the program still worked fine. When he tried a 100-digit phone number, the program crashed. The programmer had probably made an undocumented and untested decision that nobody would ever try to dial a 100-digit phone number … except Litchfield.

Having found a breaking point, Litchfield then began the interesting part of his work: Crashing a program demonstrates a fault, but exploiting that flaw shows how serious the fault is. By more experimentation, Litchfield found that the number to dial was written into the stack, the data structure that stores parameters and return addresses for embedded program calls. The dialer.exe program is treated as a program call by the operating system, so by controlling what dialer.exe overwrote, Litchfield could redirect execution to continue anywhere with any instructions he wanted. The full details of his exploitation are given in [LIT99].

This example was not the first buffer overflow, and since 1999, far more buffer overflows have been discovered. However, this example shows clearly the mind of an attacker. In this case, Litchfield was trying to improve security—he happened to be working for one of this book's authors at the time—but many more attackers work to defeat security for reasons such as those listed in Chapter 1. We now investigate sources of buffer overflow attacks, their consequences, and some countermeasures.

HARM: DESTRUCTION OF CODE AND DATA

A string overruns its assigned space or one extra element is shoved into an array; what's the big deal, you ask? To understand why buffer overflows are a major security issue, you need to understand how an operating system stores code and data.

As noted above, buffer overflows have existed almost as long as higher-level programming languages with arrays. For a long time overflows were simply a minor annoyance to programmers and users, a cause of errors and sometimes even system crashes. More recently, however, attackers have used them as vehicles to cause first a system crash and then a controlled failure with a serious security implication. The large number of security vulnerabilities based on buffer overflows shows that developers must pay more attention now to what had previously been thought to be just a minor annoyance.

Memory Allocation

Memory is a scarce but flexible resource; any memory location can hold any piece of code or data. To make managing computer memory efficient, operating systems jam one data element next to another, without regard for data type, size, content, or purpose.[1] Users and programmers seldom know, much less have any need to know, precisely which memory location a code or data item occupies.

Computers use a pointer or register known as a **program counter** that indicates the next instruction. As long as program flow is sequential, hardware bumps up the value in the program counter to point just after the current instruction as part of performing that instruction. Conditional instructions such as IF(), branch instructions such as loops (WHILE, FOR) and unconditional transfers such as GOTO or CALL divert the flow of execution, causing the hardware to put a new destination address into the program counter. Changing the program counter causes execution to transfer from the bottom of a loop back to its top for another iteration. Hardware simply fetches the byte (or bytes) at the address pointed to by the program counter and executes it as an instruction.

Instructions and data are all binary strings; only the context of use says a byte, for example, 0x41 represents the letter A, the number 65, or the instruction to move the contents of register 1 to the stack pointer. If you happen to put the data string "A" in the path of execution, it will be executed as if it were an instruction.

Not all binary data items represent valid instructions. Some do not correspond to any defined operation, for example, operation 0x78 on a machine whose instructions are all numbers between 0x01 and 0x6f. Other invalid forms attempt to use nonexistent hardware features, such as a reference to register 7 on a machine with only five hardware registers.

To help operating systems implement security, some hardware contains more than one mode of instruction: so-called privileged instructions that can be executed only when the processor is running in a protected mode. Trying to execute something that does not correspond to a valid instruction or trying to execute a privileged instruction when not in the proper mode will cause a **program fault**. When hardware generates a program fault, it stops the current thread of execution and transfers control to code that will take recovery action, such as halting the current process and returning control to the supervisor.

Code and Data

Before we can explain the real impact of buffer overflows, we need to clarify one point: Code, data, instructions, the operating system, complex data structures, user programs, strings, downloaded utility routines, hexadecimal data, decimal data, character strings, code libraries, and everything else in memory are just strings of 0s and 1s; think of it all as bytes each containing a number. The computer pays no attention to how the bytes

1. Some operating systems do separate executable code from nonexecutable data, and some hardware can provide different protection to memory addresses containing code as opposed to data. Unfortunately, however, for reasons including simple design and performance, most operating systems and hardware do not implement such a distinction. We ignore the few exceptions in this chapter because the security issue of buffer overflow applies even within a more constrained system. Designers and programmers need to be aware of buffer overflows because a program designed for use in one environment is sometimes transported to another less protected one.