Chapter 3. Program Security

In this chapter

- **Programming errors** with security implications: buffer overflows, incomplete access control
- **Malicious code**: viruses, worms, Trojan horses
- **Program development controls against malicious** code and vulnerabilities: software engineering principles and practices
- **Controls to protect against program flaws in execution**: operating system support and administrative controls

protecting programs is at the heart of computer security because **programs** constitute so much of a computing system (the **operating system**, device drivers, the **network** infrastructure, **database** management systems and other applications, executable commands on **web pages**). For now, we call all these pieces of code "**programs**."

So we need to ask **two important questions**:

- **How do we keep programs free from flaws?**
- **How do we protect computing resources against programs that contain flaws?**

security implies some degree of trust that the program enforces expected **confidentiality, integrity, and availability**.

Q: how can we look at a software component or code and assess its security? This question is, similar to the problem of **assessing** software **quality** in general. practitioners look at quantity and types of **faults** for evidence of a product's **quality**
A: One way to assess security or quality is to **ask people to name the characteristics** of software that contribute to its overall security. However, we get different answers from different people. This difference occurs because the importance of the characteristics depends on who is analyzing the software. For example, one person may decide that code is secure because it takes **too long to break** through its security controls. And someone else may decide code is secure if it has **run for a period of time with no failures**. But a third may say that **any fault in meeting security requirements** makes code insecure.

Fixing Faults : One approach to judging quality in security has been fixing faults.

computer security was based on "**penetrate and patch**," in which analysts searched for and repaired faults. Often, a top-quality "tiger team" would be convened to **test a system's security by attempting to cause it to fail.** The test was considered to be a "proof" of security.

Unfortunately, the proof became a counterexample, several serious security problems were uncovered.
The problem discovery in turn led to a rapid effort to **"patch" the system to repair or restore the security**. However, the patch efforts were largely useless, because they frequently introduced new faults. There are at least four reasons why.

- The **pressure to repair** a specific problem encouraged a narrow **focus on the fault itself and not on its context**. In particular, the analysts paid attention to the immediate cause of the failure and not to the underlying design or requirements faults.
- The **fault** often **had** non-obvious **side effects** in places other than the immediate area of the fault.
- **Fixing** one problem often **caused a failure somewhere else**, or the patch addressed the problem in only one place, not in other related places.
- The **fault could not be fixed properly** because system functionality or performance would suffer as a consequence.

IEEE Terminology for Quality
A **bug** can be a mistake in interpreting a requirement, a syntax error in a piece of *code,* or the cause of a system crash. The IEEE has suggested a standard terminology for describing bugs in our software products
When a *human* makes a mistake, called an **error**,

in performing some software activity, the *error may lead* to a **fault**, or an incorrect step, command, process, or data definition in a computer program. For example, a designer may misunderstand a requirement and create a design that does not match the actual intent of the requirements analyst and the user. This design fault can lead to other faults, such as incorrect code and an incorrect description in a user manual. <u>Thus, a single error can generate many faults, and a fault can reside in any development or maintenance product.</u>
A **failure** is a departure from the system's required behavior. It can be discovered before or after system delivery, during testing, or during operation and maintenance. Since the requirements documents can contain faults, a failure indicates that the system is not performing as required, even though it may be performing as specified.
Thus, a fault is an inside view of the system, as seen by the eyes of the developers, whereas a failure is an outside view: a problem that the user sees. Not every fault corresponds to a failure; for example, if faulty code is never executed or a particular state is never entered, then the fault will never cause the code to fail.

Types of Flaws
To aid our understanding of the problems and their prevention or correction, we can define categories that distinguish one kind of problem from another.

For example, Landwehr et al. present a taxonomy of program flaws, dividing them first into intentional and inadvertent flaws. They further divide intentional flaws into malicious and nonmalicious ones. In the taxonomy, the inadvertent flaws fall into six categories:

- **validation error** (incomplete or inconsistent): permission checks
- **domain error**: controlled access to data
- **serialization and aliasing**: program flow order
- **inadequate identification and authentication**: basis for authorization
- **boundary condition violation**: failure on first or last case
- other exploitable logic errors

Types of Malicious Code.

Code Type	**Characteristics**
Virus	Attaches itself to program & propagates copies of itself to programs
Trojan horse	Contains unexpected, additional functionality
Logic bomb	Triggers action when condition occurs
Time bomb	Triggers action when specified time occurs
Trapdoor	Allows unauthorized access to functionality
Worm	Propagates copies of itself through a network
Rabbit	Replicates itself without limit to exhaust resources

Non-malicious Program Errors

1-Buffer Overflows

A buffer overflow is the computing equivalent of trying to pour two liters of water into a one-liter pitcher: Some water is going to spill out and make a mess. And in computing, what a mess these errors have made!

Definition

A buffer (or array or string) is a space in which data can be held. A buffer resides in memory. Because memory is finite, a buffer's capacity is finite. For this reason, in many programming languages the programmer must declare the buffer's maximum size so that the compiler can set aside that amount of space.

Let us look at an example to see how buffer overflows can happen. Suppose a C language program contains the declaration:

```
char sample[10];
```

The compiler sets aside 10 bytes to store this buffer, one byte for each of the 10 elements of the array, sample[0] tHRough sample[9]. Now we execute the statement:

```
sample[10] = 'B';
```

The subscript is out of bounds (that is, it does not fall between 0 and 9), so we have a problem. The nicest outcome (from a security perspective) is for the compiler to detect the problem and mark the error during compilation. However, if the statement were

```
sample[i] = 'B';
```

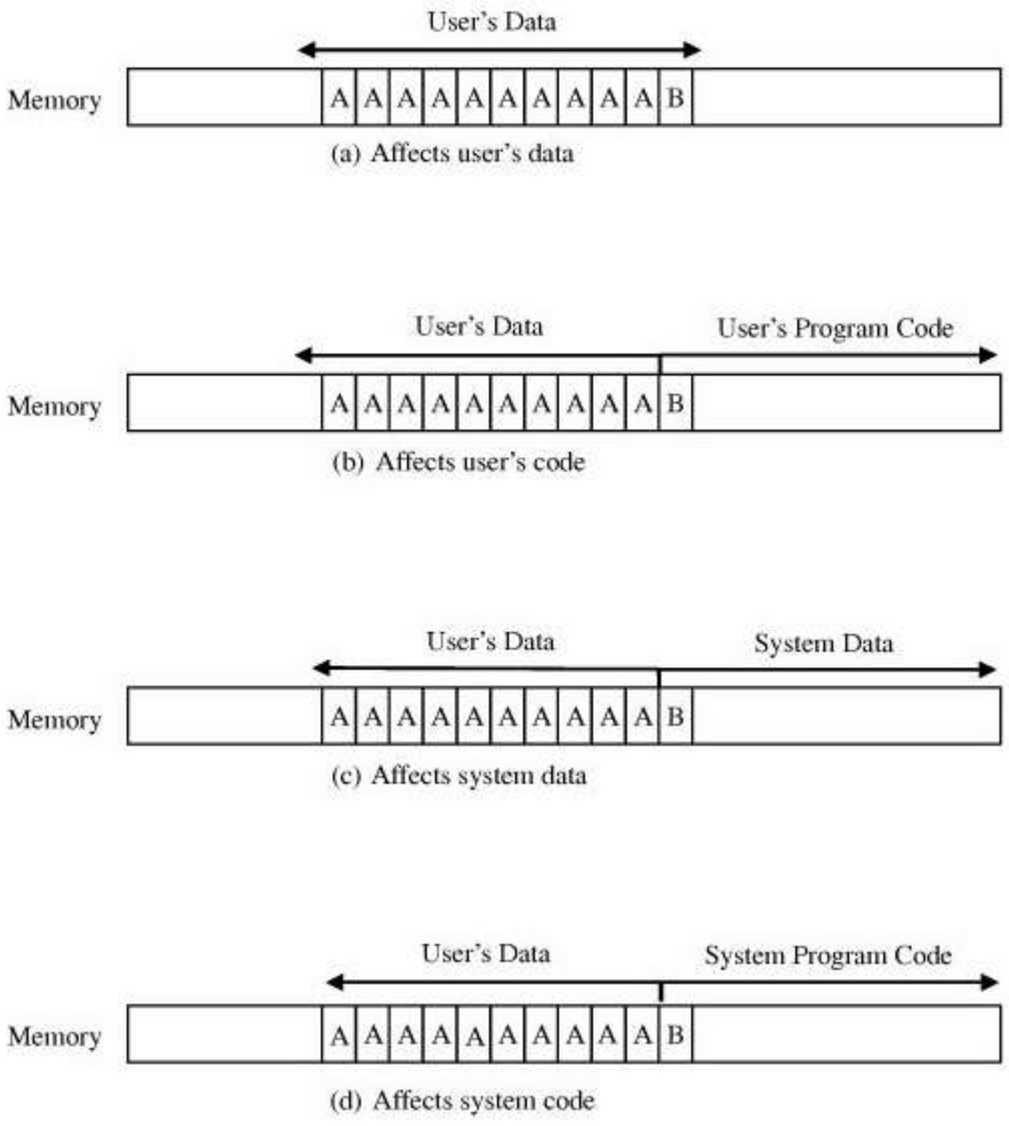
we could not identify the problem until i was set during execution to a too-big subscript. It would be useful if, during execution, the system produced an error message warning of a subscript out of bounds. Unfortunately, in some languages, buffer sizes do not have to be predefined, so there is no way to detect an out-of-bounds error. More importantly, the code needed to check each subscript against its potential maximum value takes time and space during execution, and the resources are applied to catch a problem that occurs relatively infrequently. Even if the compiler were careful in analyzing the buffer declaration and use, this same problem can be caused with pointers, for which there is no reasonable way to define a proper limit. Thus, some compilers do not generate the code to check for exceeding bounds. Let us examine this problem more closely. It is important to recognize that the potential overflow causes a serious problem only in some instances. The problem's occurrence depends on what is adjacent to the array sample. For example, suppose each of the ten elements of the array sample is filled with the letter A and the erroneous reference uses the letter B, as follows:

```
for (i=0; i<=9; i++)
     sample[i] = 'A';
sample[10] = 'B'
```

All program and data elements are in memory during execution, sharing space with the operating system, other code, and resident routines. So there are four cases to consider in deciding where the 'B' goes, as shown in Figure 3-1. If the extra character overflows into the user's data space, it simply overwrites an existing variable value (or it may be written into an as-yet unused location), perhaps affecting the program's result, but affecting no other program or data.

In the second case, the 'B' goes into the user's program area. If it overlays an already executed instruction (which will not be executed again), the user should perceive no effect. If it overlays an instruction that is not yet executed, the machine will try to execute an instruction with operation code 0x42, the internal code for the character 'B'. If there is no instruction with operation code 0x42, the system will halt on an illegal instruction exception. Otherwise, the machine will use subsequent bytes as if they were the rest of the instruction, with success or failure depending on the meaning of the contents. Again, only the user is likely to experience an effect.
The most interesting cases occur when the system owns the space immediately after the array that overflows. Spilling over into system data or code areas produces similar results to those for the user's space: computing with a faulty value or trying to execute an improper operation.

Figure 3-1. Places Where a Buffer Can Overflow.



Security Implication

In this section we consider program flaws from unintentional or nonmalicious causes. Remember, however, that even if a flaw came from an honest mistake, the flaw can still cause serious harm. A malicious attacker can exploit these flaws.

Let us suppose that a malicious person understands the damage that can be done by a buffer overflow; that is, we are dealing with more than simply a normal, errant programmer. The malicious programmer looks at the four cases illustrated in Figure 3-1 and thinks deviously about the last two: What data values could the attacker insert just after the buffer to cause mischief or damage, and what planned instruction codes could the system be forced to execute? There are many possible answers, some of which are more malevolent than others. Here, we present two buffer overflow attacks that are used frequently. (See [ALE96] for more details.)

First, the attacker may replace code in the system space. Remember that every program is invoked by the operating system and that the operating system may run with higher privileges than those of a regular program. Thus, if the attacker can gain control by masquerading as the operating system, the attacker can execute many commands in a powerful role. Therefore, by replacing a few instructions right after returning from his or her own procedure, the attacker regains control from the operating system, possibly with raised privileges. If the buffer overflows into system code space, the attacker merely inserts overflow data that correspond to the machine code for instructions.

On the other hand, the attacker may make use of the stack pointer or the return register. Subprocedure calls are handled with a stack, a data structure in which the most recent item inserted is the next one removed (last arrived, first served). This structure works well because procedure calls can be nested, with each return causing control to transfer back to the immediately preceding routine at its point of execution. Each time a procedure is called, its parameters, the return address (the address immediately after its call), and other local values are pushed onto a stack. An old stack pointer is also pushed onto the stack, and a stack pointer register is reloaded with the address of these new values. Control is then transferred to the subprocedure. As the subprocedure executes, it fetches parameters that it finds by using the address pointed to by the stack pointer. Typically, the stack pointer is a register in the processor. Therefore, by causing an overflow into the stack, the attacker can change either the old stack pointer (changing the context for the calling procedure) or the return address (causing control to transfer where the attacker wants when the subprocedure returns). Changing the context or return address allows the attacker to redirect execution to a block of code the attacker wants.

As noted earlier, buffer overflows have existed almost as long as higher-level programming languages with arrays. For a long time they were simply a minor annoyance to programmers and users, a cause of errors and sometimes even system crashes. Rather recently, 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.

2-Incomplete Mediation

Incomplete mediation is another security problem that has been with us for decades. Attackers are exploiting it to cause security problems.

Definition

Consider the example of the previous section:
http://www.somesite.com/subpage/userinput.asp?parm1=(808)555-1212
&parm2=2009Jan17

Security Implication

Incomplete mediation is easy to exploit, but it has been exercised less often than buffer overflows. Nevertheless, unchecked data values represent a serious potential vulnerability.

To demonstrate this flaw's security implications, we use a real example; only the name of the vendor has been changed to protect the guilty. Things, Inc., was a very large, international vendor of consumer products, called Objects. The company was ready to sell its Objects through a web site, using what appeared to be a standard ecommerce application. The management at Things decided to let some of its in-house developers produce the web site so that its customers could order Objects directly from the web.

So far, so good; everything in the parameter passage looks correct. But this procedure leaves the parameter statement open for malicious tampering. Things should not need to pass the price of the items back to itself as an input parameter; presumably Things knows how much its Objects cost, and they are unlikely to change dramatically since the time the price was quoted a few screens earlier.
A malicious attacker may decide to exploit this peculiarity by supplying instead the following URL, where the price has been reduced from $205 to $25:
http://www.things.com/order.asp?custID=101&part=555A&qy=20&price =1&ship=boat&shipcost=5&total=25

3-Time-of-Check to Time-of-Use Errors
The third programming flaw we investigate involves synchronization. To improve efficiency, modern processors and operating systems usually change the order in which instructions and procedures are executed. In particular, instructions that appear to be adjacent may not actually be executed immediately after each other, either because of intentionally changed order or because of the effects of other processes in concurrent execution.
Definition
Access control is a fundamental part of computer security; we want to make sure that only those who should access an object are allowed that access. (We explore the access control mechanisms in operating systems in greater detail in Chapter 4.) Every requested access must be governed by an access policy stating who is allowed access to what; then the request must be mediated by an access-policy-enforcement agent. But an incomplete mediation problem occurs when access is not checked universally. The **time-of-check to time-of-use** (TOCTTOU) flaw concerns mediation that is performed with a "bait and switch" in the middle. It is also known as a serialization or synchronization flaw.
To understand the nature of this flaw, consider a person's buying a sculpture that costs $100. The buyer removes five $20 bills from a wallet, carefully counts them in front of the seller, and lays them on the table. Then the seller turns around to write a receipt. While the seller's back is turned, the buyer takes back one $20 bill. When the seller turns around, the buyer hands over the stack of bills, takes the receipt, and leaves with the sculpture. Between the time the security was checked (counting the bills) and the access (exchanging the sculpture for the bills), a condition changed: What was checked is no longer valid when the object (that is, the sculpture) is accessed.