Part IV

Software

Chapter 11

Software Flaws and Malware

If automobiles had followed the same development cycle as the computer, a Rolls-Royce would today cost $100, get a million miles per gallon, and explode once a year, killing everyone inside.
— Robert X. Cringely

My software never has bugs. It just develops random features.
— Anonymous

11.1 Introduction

Why is software an important security topic? Is it really on par with crypto, access control, and protocols? For one thing, virtually all of information security is implemented in software. If your software is subject to attack, all of your other security mechanisms are vulnerable. In effect, software is the foundation on which all other security mechanisms rest. We'll see that software provides a poor foundation on which to build security—comparable to building your house on quicksand.[1]

In this chapter, we'll discuss several software security issues. First, we consider unintentional software flaws that can cause security problems [183]. Then we consider malicious software, or malware, which is intentionally designed to be bad. We'll also discuss the future of malware, and we'll mention a few other types of software-based attacks.

Software is a big subject, so we continue with software-related security topics in the next two chapters. Even with three chapters worth of material we can, as usual, do little more than scratch the surface.

[1]Or, in an analogy that is much closer to your fearless author's heart, it's like building a house on a hillside in earthquake country.

11.2 Software Flaws

Bad software is everywhere [143]. For example, the NASA Mars Lander, which cost $165 million, crashed into Mars due to a software error related to converting between English and metric units of measure [150]. Another infamous example is the Denver airport baggage handling system. Bugs in this software delayed the airport opening by 11 months at a cost of more than $1 million per day [122].[2] Software failures also plagued the MV-22 Osprey, an advanced military aircraft—lives were lost due to this faulty software [178]. Attacks on smart electric meters, which have the potential to incapacitate the power grid, have been blamed on buggy software [127]. There are many many more examples of such problems.

In this section, we're interested in the security implications of software flaws. Since faulty software is everywhere, it shouldn't be surprising that the bad guys have found ways to take advantage of this situation.

Normal users find software bugs and flaws more or less by accident. Such users hate buggy software, but out of necessity, they've learned to live with it. Users are surprisingly good at making bad software work.

Attackers, on the other hand, look at buggy software as an opportunity, not a problem. They actively search for bugs and flaws in software, and they like bad software. Attackers try to make software misbehave, and flaws can prove very useful in this regard. We'll see that buggy software is at the core of many (if not most) attacks.

It's generally accepted among computer security professionals that complexity is the enemy of security [74], and modern software is extremely complex. In fact, the complexity of software has far outstripped the abilities of humans to manage the complexity. The number of lines of code (LOC) in a piece of software is a crude measure of its complexity—the more lines of code, the more complex. The numbers in Table 11.1 highlight the extreme complexity of large-scale software projects.

Conservative estimates place the number of bugs in commercial software at about 0.5 per 1,000 LOC [317]. A typical computer might have 3,000 executable files, each of which contains the equivalent of, perhaps, 100,000 LOC, on average. Then, on average, each executable has 50 bugs, which implies about 150,000 bugs living in a single computer.

If we extend this calculation to a a medium-sized corporate network with 30,000 nodes, we'd expect to find about 4.5 billion bugs in the network. Of

[2]The automated baggage handling system proved to be an "unmitigated failure" [87] and it was ultimately abandoned in 2005. As an aside, it's interesting to note that this expensive failure was only the tip of the iceberg in terms of cost overruns and delays for the overall airport project. And, you might be wondering, what happened to the person responsible for this colossal waste of taxpayer money? He was promoted to U.S. Secretary of Transportation [170].

Table 11.1: Approximate Lines of Code

System	LOC
Netscape	17 million
Space shuttle	10 million
Linux kernel 2.6.0	5 million
Windows XP	40 million
Mac OS X 10.4	86 million
Boeing 777	7 million

course, many of these bugs would be duplicates, but 4.5 billion is still a staggering number.

Now suppose that only 10% of bugs are security critical and that only 10% of these are remotely exploitable. Then our typical corporate network "only" has 4.5 million serious security flaws that are directly attributable to bad software!

The arithmetic of bug counting is good news for the bad guys and very bad news for the good guys. We'll return to this topic later, but the crucial point is that we are not going to eliminate software security flaws any time soon—if ever. We'll discuss ways to reduce the number and severity of flaws, but many flaws will inevitably remain. The best we can realistically hope for is to effectively manage the security risk created by buggy and complex software. In almost any real-world situation, absolute security is often unobtainable, and software is definitely no exception.[3]

In this section, we'll focus on program flaws. These are unintentional software bugs that can have security implications. We'll consider the following specific classes of flaws.

- Buffer overflow
- Race conditions
- Incomplete mediation

After covering these unintentional flaws, we'll turn our attention to malicious software, or malware. Recall that malware is designed to do bad things.

A programming mistake, or bug, is an *error*. When a program with an error is executed, the error might (or might not) cause the program to reach

[3]One possible exception is cryptography—if you use strong crypto, and use it correctly, you are as close to absolutely secure as you will ever be. However, crypto is usually only one part of a security system, so even if your crypto is perfect, many vulnerabilities will likely remain. Unfortunately, people often equate crypto with information security, which leads some to mistakenly expect absolute security.

an incorrect internal state, which is known as a *fault*. A fault might (or might not) cause the system to depart from its expected behavior, which is a *failure* [235]. In other words, an error is a human-created bug, while a fault is internal to the software, and a failure is externally observable.

For example, the C program in Table 11.2 has an error, since `buffer[20]` has not been allocated. This error might cause a fault, where the program reaches an incorrect internal state. If a fault occurs, it might lead to a failure, where the program behaves incorrectly (e.g., the program crashes). Whether a fault occurs, and whether this leads to a failure, depends on what resides in the memory location where `buffer[20]` is written. If that particular memory location is not used for anything important, the program might execute normally, which makes debugging challenging.

Table 11.2: A Flawed Program

```
int main(){
    int buffer[10];
    buffer[20] = 37;}
```

Distinguishing between errors, faults, and failures is a little too pedantic for our purposes. So, for the remainder of this section, we use the term *flaw* as a synonym for all three. The severity should be apparent from context.

One of the primary goals in software engineering is to ensure that a program does what it's supposed to do. However, for software to be secure, a much higher standard is required—secure software software must do what it's supposed to do *and nothing more* [317]. It's difficult enough just trying to ensure that a program does what it's supposed to do. Trying to ensure that a program does "nothing more" is asking for a lot more.

Next, we'll consider three specific types of program flaws that can create significant security vulnerabilities. The first of these is the infamous stack-based *buffer overflow*, also known as *smashing the stack*. Stack smashing has been called the attack of the decade for the 1990s [14] and it's likely to be the attack of the decade for the current decade, regardless of which decade happens to be current. There are several variants of the buffer overflow attack we discuss. These variants are considered in problems at the end of the chapter.

The second class of software flaws we'll consider are *race conditions*. These are common, but generally much more difficult to exploit than buffer overflows. The third major software vulnerability that we consider is *incomplete mediation*. This is the flaw that often makes buffer overflow conditions exploitable. There are other types of software flaws, but these three represent the most common sources of problems.

11.2.1 Buffer Overflow

Alice says, "My cup runneth over, what a mess."
Trudy says, "Alice's cup runneth over, what a blessing."
— Anonymous

Before we discuss buffer overflow attacks in detail, let's consider a scenario where such an attack might arise. Suppose that a Web form asks the user to enter data, such as name, age, date of birth, and so on. The entered information is then sent to a server and the server writes the data entered in the "name" field to a buffer[4] that can hold N characters. If the server software does not verify that the length of the name is at most N characters, then a buffer overflow might occur.

It's reasonably likely that any overflowing data will overwrite something important and cause the computer to crash (or thread to die). If so, Trudy might be able to use this flaw to launch a denial of service (DoS) attack. While this could be a serious issue, we'll see that a little bit of cleverness on Trudy's part can turn a buffer overflow into a much more devastating attack. Specifically, it is sometimes possible for Trudy to execute code of her choosing on the affected machine. It's remarkable that a common programming bug can lead to such an outcome.

Consider again the C source code that appears in Table 11.2. When this code is executed, a buffer overflow occurs. The severity of this particular buffer overflow depends on what resided in memory at the location corresponding to `buffer[20]` before it was overwritten. The buffer overflow might overwrite user data or code, or it could overwrite system data or code, or it might overwrite unused space.

Consider, for example, software that is used for authentication. Ultimately, the authentication decision resides in a single bit. If a buffer overflow overwrites this authentication bit, then Trudy can authenticate herself as, say, Alice. This situation is illustrated in Figure 11.1, where the "F" in the position of the boolean flag indicates failed authentication.

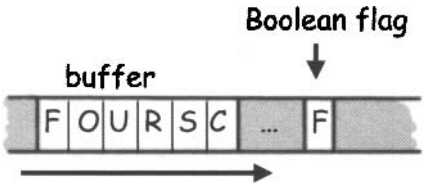


Figure 11.1: Buffer and a Boolean Flag

[4]Why is it a "buffer" and not an "array"? Obviously, it's because we're talking about buffer overflow, not array overflow...

If a buffer overflow overwrites the memory position where the boolean flag is stored, Trudy can overwrite "F" (i.e., a 0 bit) with "T" (i.e., a 1 bit), and the software will believe that Trudy has been authenticated. This attack is illustrated in Figure 11.2.

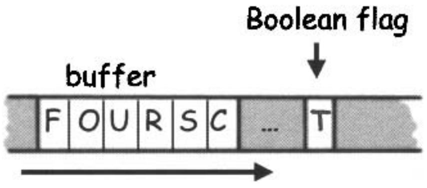


Figure 11.2: Simple Buffer Overflow

Before we can discuss the more sophisticated forms of the buffer overflow attack, we give a quick overview of memory organization for a typical modern processor. A simplified view of memory—which is sufficient for our purposes—appears in Figure 11.3. The *text* section is for code, while the *data* section holds static variables. The *heap* is for dynamic data, while the *stack* can be viewed as "scratch paper" for the processor. For example, dynamic local variables, parameters to functions, and the return address of a function call are all stored on the stack. The *stack pointer*, or SP, indicates the top of the stack. Notice that the stack grows up from the bottom in Figure 11.3, while the heap grows down.

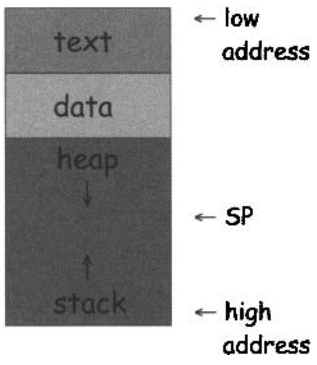


Figure 11.3: Memory Organization

11.2.1.1 Smashing the Stack

Smashing the stack refers to a particularly devastating attack that relies on a buffer overflow. For a stack smashing attack, Trudy is interested in the stack during a function call. To see how the stack is used during a function call, consider the simple example in Table 11.3.

Table 11.3: Code Example

```
void func(int a, int b){
   char buffer[10];
}
void main(){
   func(1,2);
}
```

When the function `func` in Table 11.3 is called, the values that are pushed onto the stack appear in Figure 11.4. Here, the stack is being used to provide space for the array `buffer` while the function executes. The stack also holds the return address where control will resume after the function finishes executing. Note that `buffer` is positioned above the return address on the stack, that is, `buffer` is pushed onto the stack after the return address. As a result, if the buffer overflows, the overflowing data will overwrite the return address. This is the crucial fact that makes the buffer overflow attack so lethal.

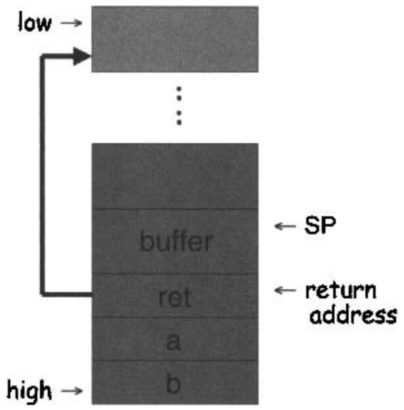


Figure 11.4: Stack Example

The `buffer` in Table 11.3 holds 10 characters. What happens if we put more than 10 characters into `buffer`? The buffer will overflow, analogous to the way that a 5-gallon gas tank will overflow if we try to add 10 gallons of gas. In both cases, the overflow will likely cause a mess. In the buffer overflow case, Figure 11.4 shows that the buffer will overflow into the space where the return address is located, thereby "smashing" the stack. Our assumption here is that Trudy has control over the bits that go into `buffer` (e.g., the "name" field in a Web form).

If Trudy overflows `buffer` so that the return address is overwritten with random bits, the program will jump to a random memory location when the function has finished executing. In this case, which is illustrated in Figure 11.5, the most likely outcome is that the program crashes.

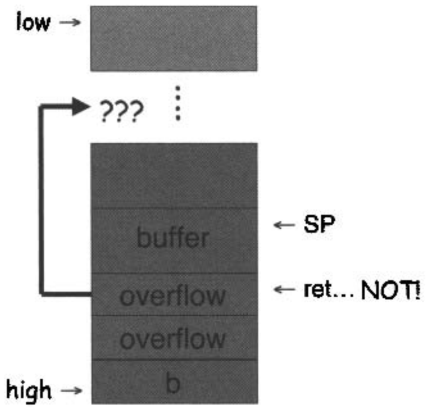


Figure 11.5: Buffer Overflow Causes a Problem

Trudy might be satisfied with simply crashing a program. But Trudy is clever enough to realize that there's much more potential to cause trouble in this situation. Since Trudy can overwrite the return address with a random address, can she also overwrite it with a specific address of her choosing? Often, the answer is yes. If so, what specific address might Trudy want to choose?

With some trial and error, Trudy can probably overwrite the return address with the address of the start of `buffer`. Then the program will try to "execute" the data stored in the buffer. Why might this be useful to Trudy? Recall that Trudy can choose the data that goes into the buffer. So, if Trudy can fill the buffer with "data" that is valid executable code, Trudy can execute this code on the victim's machine. The bottom line is that Trudy gets to execute code of her choosing on the victim's computer. This has to be bad for security. This clever version of the stack smashing attack is illustrated in Figure 11.6.

It's worth reflecting on the buffer overflow attack illustrated in Figure 11.6. Due to an unintentional programming error, Trudy can, in some cases, overwrite the return address, causing code of her choosing to execute on a remote machine. The security implications of such an attack are mind-boggling.

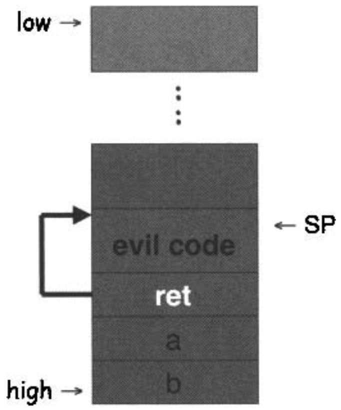


Figure 11.6: Evil Buffer Overflow

From Trudy's perspective, there are a couple of difficulties with this stack smashing attack. First, Trudy may not know the precise address of the evil code she has inserted into `buffer`, and second, she may not know the precise location of the return address on the stack. Neither of these presents an insurmountable obstacle.

Two simple tricks make a buffer overflow attack much easier to mount. For one, Trudy can precede the injected evil code with a `NOP` "landing pad" and, for another, she can insert the desired return address repeatedly. Then, if any of the multiple return addresses overwrite the actual return address, execution will jump to the specified address. And if this specified address lands on any of the inserted `NOP`s, the evil code will be executed immediately after the last `NOP` in the landing pad. This improved stack smashing attack is illustrated in Figure 11.7.

For a buffer overflow attack to succeed, obviously the program must contain a buffer overflow flaw. Not all buffer overflows are exploitable, but those that are enable Trudy to inject code into the system. That is, if Trudy finds an exploitable buffer overflow, she can execute code of her choosing on the affected system. Trudy will probably have some work to do to develop a useful attack, but it certainly can be done. And there are plenty of sources available online to help Trudy hone her skills—the standard reference is [8].

11.2.1.2 Stack Smashing Example

In this section, we'll examine code that contains an exploitable buffer overflow and we'll demonstrate an attack. Of course, we'll be working from Trudy's perspective.

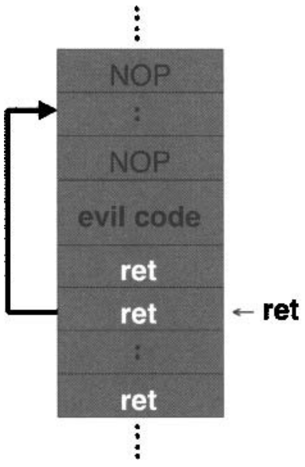
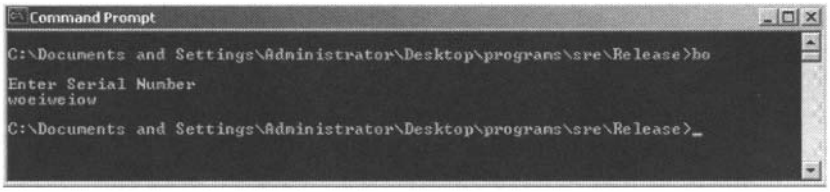


Figure 11.7: Improved Evil Buffer Overflow

Suppose that Trudy is confronted with a program that asks for a serial number—a serial number that Trudy doesn't know. Trudy wants to use the program, but she's too cheap to pay money to obtain a valid serial number.[5] Trudy does not have access to the source code, but she does possess the executable.

When Trudy runs the program and enters an incorrect serial number, the program halts without providing any further information, as indicated in Figure 11.8. Trudy proceeds to try a few different serial numbers, but, as expected, she is unable to guess the correct serial number.



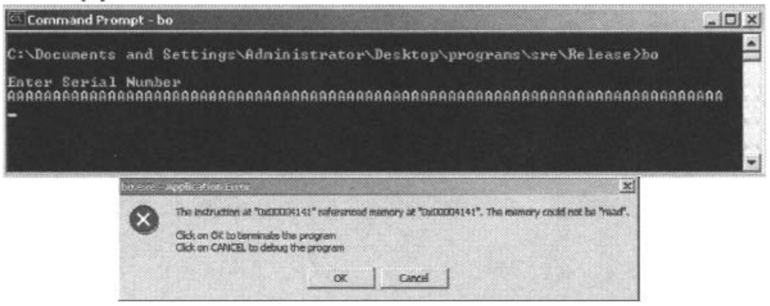
```
C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>bo
Enter Serial Number
woeiweiow
C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>_
```

Figure 11.8: Serial Number Program

Trudy then tries entering unusual input values to see how the program reacts. She is hoping that the program will misbehave in some way and that she might have a chance of exploiting the incorrect behavior. Trudy realizes she's in luck when she observes the result in Figure 11.9. This result indicates

[5]In the real world, Trudy would be wise to Google for a serial number. But let's assume that Trudy can't find a valid serial number online.

that the program has a buffer overflow. Note that 0x41 is the ASCII code for the character "A." By carefully examining the error message, Trudy realizes that she has overwritten exactly two bytes of the return address with the character A.



```
C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>bo

Enter Serial Number
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

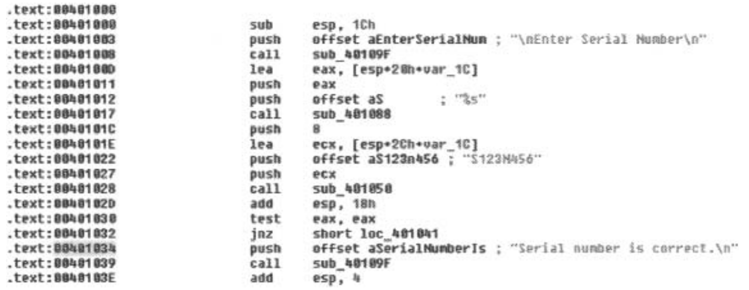
bo.exe - Application Error

The instruction at "0x00004141" referenced memory at "0x00004141". The memory could not be "read".

Click on OK to terminate the program
Click on CANCEL to debug the program

Figure 11.9: Buffer Overflow in Serial Number Program

Trudy then disassembles[6] the exe file and obtains the assembly code that appears in Figure 11.10. The significant information in this code is the "Serial number is correct" string, which appears at address 0x401034. If Trudy can overwrite the return address with the address 0x401034, then the program will jump to "Serial number is correct" and she will have obtained access to the code, without having any knowledge of the correct serial number.



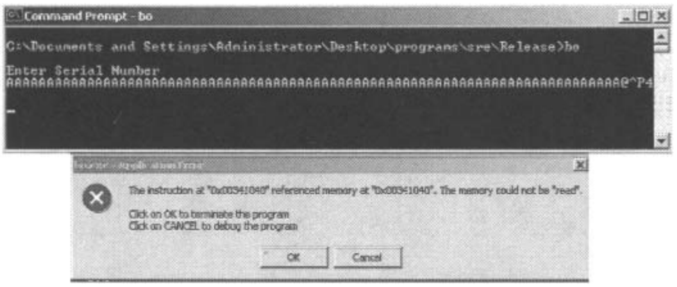
```
.text:00401000
.text:00401000          sub     esp, 1Ch
.text:00401003          push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008          call    sub_40109F
.text:0040100D          lea     eax, [esp+20h+var_1C]
.text:00401011          push    eax
.text:00401012          push    offset aS       ; "%s"
.text:00401017          call    sub_401088
.text:0040101C          push    8
.text:0040101E          lea     ecx, [esp+2Ch+var_1C]
.text:00401022          push    offset aS123n456 ; "S123N456"
.text:00401027          push    ecx
.text:00401028          call    sub_401050
.text:0040102D          add     esp, 18h
.text:00401030          test    eax, eax
.text:00401032          jnz     short loc_401041
.text:00401034          push    offset aSerialNumberIs ; "Serial number is correct.\n"
.text:00401039          call    sub_40109F
.text:0040103E          add     esp, 4
```

Figure 11.10: Disassembled Serial Number Program

But Trudy can't directly enter a hex address for the serial number, since the input is interpreted as ASCII text. Trudy consults an ASCII table where she finds that 0x401034 is "@^P4" in ASCII, where "^P" is control-P. Confident of success, Trudy starts the program, then enters just enough characters

[6]We'll have more to say about disassemblers in the next chapter when we cover software reverse engineering.

so that she is poised to overwrite the return address, and then she enters "@^P4." To her surprise, Trudy obtains the results in Figure 11.11.

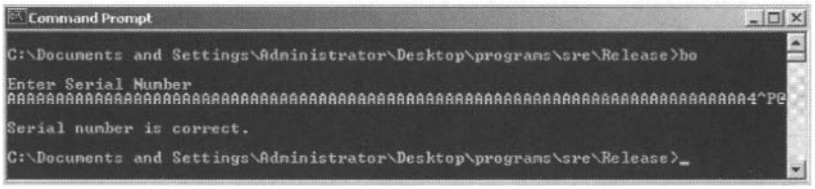


```
C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>bo

Enter Serial Number
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA@^P4

_
```

```
The instruction at "0x00341040" referenced memory at "0x00341040". The memory could not be "read".

Click on OK to terminate the program
Click on CANCEL to debug the program
```

Figure 11.11: Failed Buffer Overflow Attack

A careful examination of the error message shows that the address where the error arose was `0x341040`. Apparently, Trudy caused the program to jump to this address instead of her intended address of `0x401034`. Trudy notices that the intended address and the actual address are byte-reversed. The problem here is that the machine Trudy is dealing with uses the little endian convention, so that the low-order byte is first and the high-order byte comes last. That is, the address that Trudy wants, namely, `0x401034`, is stored internally as `0x341040`. So Trudy changes her attack slightly and overwrites the return address with `0x341040`, which in ASCII is "4^P@." With this change, Trudy is successful, as shown in Figure 11.12.



```
C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>bo

Enter Serial Number
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA4^P@

Serial number is correct.

C:\Documents and Settings\Administrator\Desktop\programs\sre\Release>_
```

Figure 11.12: Successful Buffer Overflow Attack

The point of this example is that without knowledge of the serial number, and without access to the source code, Trudy was able to break the security of the software. The only tool she used was a disassembler to determine the address that she needed to use to overwrite the return address. In principle, this address could be found by trial and error, although that would be tedious, at best. If Trudy has the executable in her possession, she would be foolish not to employ a disassembler—and Trudy is no fool.

For the sake of completeness, we provide the C source code, `bo.c`, corresponding to the executable, `bo.exe`. This source code appears in Table 11.4.

Table 11.4: Source Code for Serial Number Example

```
main()
{
   char in[75];
   printf("\nEnter Serial Number\n");
   scanf("%s", in);
   if(!strncmp(in, "S123N456", 8))
   {
      printf("Serial number is correct.\n");
   }
}
```

Again, Trudy was able to complete her buffer overflow attack without access to the source code in Table 11.4. We provide the source code here for reference.

Finally, note that in this buffer overflow example, Trudy did not execute code on the stack. Instead, she simply overwrote the return address, which caused the program to execute code that already existed at the specified address. That is, no code injection was employed, which greatly simplfies the attack. This version of stack smashing is usually referred to as a *return-to-libc* attack.

11.2.1.3 Stack Smashing Prevention

There are several possible ways to prevent stack smashing attacks. One approach is to eliminate all buffer overflows from software. However, this is more difficult than it sounds and even if we eliminate all such bugs from new software, there is a huge base of existing software that is riddled with buffer overflows.

Another option is to detect buffer overflows as they occur and respond accordingly. Some programming languages do this automatically. Yet another option is to not allow code to execute on the stack. Finally, if we randomize the location where code is loaded into memory, then the attacker cannot know the address where the `buffer` or other code is located, which would prevent most buffer overflow attacks. In this section, we'll briefly discuss these various options.

An easy way to minimize the damage caused by many stack-based buffer overflows is to make the stack non-executable, that is, do not allow code to execute on the stack. Some hardware (and many operating systems) support this *no execute*, or NX bit [129]. Using the NX bit, memory can be flagged so that code can't execute in specified locations. In this way the stack (as well as the heap and data sections) can be protected from many buffer overflow attacks. Recent versions of Microsoft Windows support the NX bit [311].

As the NX approach becomes more widely deployed and used, we should see a decline in the number and severity of buffer overflow attacks. However, NX will not prevent all buffer overflow attacks. For example, the return-to-libc attack discussed in the previous section would not be affected. For more information on NX and its security implications, see [173].

Using safe programming languages such as Java or C# will eliminate most buffer overflows at the source. These languages are safe because at runtime they automatically check that all memory accesses are within the declared array bounds. Of course, there is a performance penalty for such checking, and for that reason much code will continue to be written in C, particularly for applications destined for resource-constrained devices. In contrast to these safe languages, there are several C functions that are known to be unsafe and these functions are the source of the vast majority of buffer overflow attacks. There are safer alternatives to all of the unsafe C functions, so the unsafe functions should never be used—see the problems at the end of the chapter for more details.

Runtime stack checking can be used to prevent stack smashing attacks. In this approach, when the return address is popped off of the stack, it's checked to verify that it hasn't changed. This can be accomplished by pushing a special value onto the stack immediately after the return address. Then when Trudy attempts to overwrite the return address, she must first overwrite this special value, which provides a means for detecting the attack. This special value is usually known as a *canary*, in reference to the coal miner's canary.[7] The use of a canary for stack smashing detection is illustrated in Figure 11.13.

Note that if Trudy can overwrite an anti-stack-smashing canary with itself, then her attack will go undetected. Can we prevent the canary from being overwritten with itself?

A canary can be a constant, or a value that depends on the return address. A specific constant that is sometimes used is `0x000aff0d`. This constant includes `0x00` as the first byte since this is the string terminating byte. Any string that overflows a buffer and includes `0x00` will be terminated at that point and no more of the stack will be overwritten. Consequently, an attacker can't use a string input to overwrite the constant `0x000aff0d` with itself, and any other value that overwrites the canary will be detected. The other bytes in this constant serve to prevent other types of buffer overflow attacks.

[7]Coal miners would take a canary with them underground into the mine. If the canary died, the coal miners knew there was a problem with the air and they needed to get out of the mine as soon as possible.

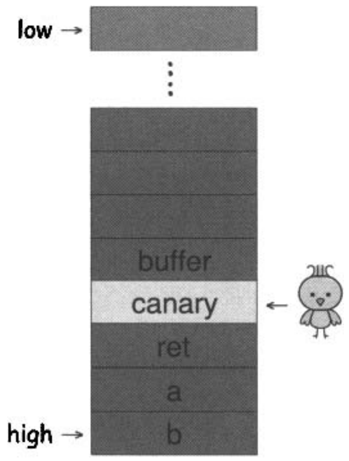


Figure 11.13: Canary

Microsoft recently added a canary feature to its C++ compiler based on the approach discussed in [246]. Any program compiled with the /GS compiler flag will use a canary—or, in Microsoft-speak, a "security cookie"—to detect buffer overflows at runtime. But the initial Microsoft implementation was apparently flawed. When the canary died, the program passed control to a user-supplied handler function. It was discovered that an attacker could specify this handler function, thereby executing arbitrary code on the victim machine [245], although the severity of this attack was disputed by Microsoft [187]. Assuming the claimed attack was valid, then all buffer overflows compiled under the /GS option were exploitable, even those that would not have been exploitable without the /GS option. In other words, the cure was worse than the disease.

Another option for minimizing the effectiveness of buffer overflow attacks is Address Space Layout Randomization, or ASLR [105]. This technique is used in recent Windows operating systems and several other modern OSs. ASLR relies on the fact that buffer overflow attacks are fairly delicate. That is, to execute code on the stack, Trudy usually overwrites the return address with a hard-coded specific address that causes execution to jump to the specified location. When ASLR is used, programs are loaded into more or less random locations in memory, so that any address that Trudy has hard-coded into her attack is only likely to be correct a small percentage of the time. Then Trudy's attack will only succeed a correspondingly small percentage of the time.

However, in practice, only a relatively small number of "random" layouts are used. Vista, for example, uses 256 distinct layouts and, consequently, a given buffer overflow attacks should have a natural success probability of about 1/256. However, due to a weakness in the implementation, Vista does not choose from these 256 possible layouts uniformly, which results in a significantly greater chance of success for a clever attacker [324]. In addition, a so-called de-randomization attack on certain specific ASLR implementations is discussed in [263].

11.2.1.4 Buffer Overflow: The Last Word

Buffer overflow was unquestionably the attack of the decade for each of the past several decades. For example, buffer overflow has been the enabling vulnerability in many major malware outbreaks. This, in spite of the fact that buffer overflow attacks have been well known since the 1970s, and it's possible to prevent most such attacks by using the NX bit approach and/or safe programming languages and/or ASLR. Even with an unsafe language such as C, buffer overflow attacks can be greatly reduced by using the safer versions of the unsafe functions.

Can we hope to relegate buffer overflow attacks to the scrapheap of history? Developers must be educated, and tools for preventing and detecting buffer overflow conditions must be used. If it's available on a given platform, the NX bit should certainly be employed and ASLR is a very promising technology. Unfortunately, buffer overflows will remain a problem for the foreseeable future because of the large amount of legacy code and older machines that will continue to be in service.

11.2.2 Incomplete Mediation

The C function `strcpy(buffer, input)` copies the contents of the input string `input` to the array `buffer`. As we discovered above, a buffer overflow will occur if the length of `input` is greater than the length of `buffer`. To prevent such a buffer overflow, the program must validate the input by checking the length of `input` before attempting to write it to `buffer`. Failure to do so is an example of *incomplete mediation*.

As a somewhat more subtle example, consider data that is input to a Web form. Such data is often transferred to the server by embedding it in a URL, so that's the method we'll employ here. Suppose the input is validated on the client before constructing the required URL.

For example, consider the following URL:

```
http://www.things.com/orders/final&custID=112&
    num=55A&qty=20&price=10&shipping=5&total=205
```

On the server, this URL is interpreted to mean that the customer with ID number 112 has ordered 20 of item number 55, at a cost of $10 each, with a $5 shipping charge, giving a total cost of $205. Since the input was checked on the client, the developer of the server software believes it would be wasted effort to check it again on the server.

However, instead of using the client software, Trudy can directly send a URL to the server. Suppose Trudy sends the following URL to the server:

```
http://www.things.com/orders/final&custID=112&
    num=55A&qty=20&price=10&shipping=5&total=25
```

If the server doesn't bother to validate the input, Trudy can obtain the same order as above, but for the bargain basement price of $25 instead of the legitimate price of $205.

Recent research [79] revealed numerous buffer overflows in the Linux kernel, and most of these were due to incomplete mediation. This is perhaps somewhat surprising since the Linux kernel is usually considered to be very good software. After all, it is open source, so anyone can look for flaws in the code (we'll have more to say about this in the next chapter) and it is the kernel, so it must have been written by experienced programmers. If these software flaws are common in such code, they are undoubtedly more common in most other code.

There are tools available to help find likely cases of incomplete mediation. These tools should be more widely used, but they are not a cure-all since this problem can be subtle, and therefore difficult to detect automatically. As with most security tools, these tools can also be useful for the bad guys.

11.2.3 Race Conditions

Ideally, security processes should be *atomic*, that is, they should occur all at once. So-called race conditions can arise when a security-critical process occurs in stages. In such cases, an attacker may be able to make a change between the stages and thereby break the security. The term race condition refers to a "race" between the attacker and the next stage of the process, although it's not so much a race as a matter of careful timing for the attacker.

The race condition that we'll consider occurs in an outdated version of the Unix command `mkdir`, which creates a new directory. With this version of `mkdir`, the directory is created in stages—there is a stage that determines authorization followed by a stage that transfers ownership. If Trudy can make a change after the authorization stage but before the transfer of ownership, then she can, for example, become the owner of some directory that she should not be able to access.

The way that this version of `mkdir` is supposed to work is illustrated in Figure 11.14. Note that `mkdir` is not atomic and that is the source of the race condition.

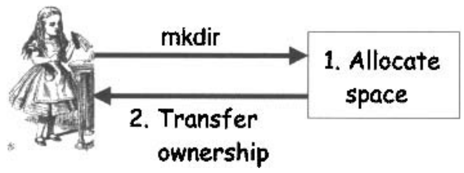


Figure 11.14: How `mkdir` is Supposed to Work

Trudy can exploit this particular `mkdir` race condition if she can somehow implement the attack that is illustrated in Figure 11.15. In this attack scenario, after the space for the new directory is allocated, a link is established from the the password file (which Trudy is not authorized to access) to this newly created space, before ownership of the new directory is transferred to Trudy. Note that this attack is not really a race, but instead it requires careful (or lucky) timing by Trudy.

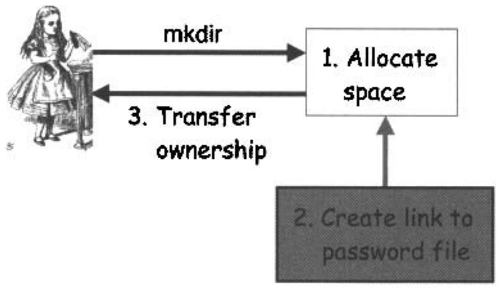


Figure 11.15: Attack on `mkdir` Race Condition

Today, race conditions are probably fairly common and with the trend towards increased parallelism, they are sure to become even more prevalent. However, real-world attacks based on race conditions are rare—attackers clearly favor buffer overflows.

Why are attacks based on race conditions a rarity? For one thing, exploiting a race condition requires careful timing. In addition, each race condition is unique, so there is no standard formula for such an attack. In comparison to, say, buffer overflow attacks, race conditions are certainly more difficult to exploit. Consequently, as of today buffer overflows are the low hanging fruit and are therefore favored by attackers. However, if the number of buffer overflows is reduced, or buffer overflows are made sufficiently difficult to exploit, it's a safe bet that we will see a corresponding increase in attacks based on race conditions. This is yet another illustration of Stamp's Principle: there is job security in security.

11.3 Malware

Solicitations malefactors!
— Plankton

In this section, we'll discuss software that is designed to break security. Since such software is malicious in its intent, it goes by the name of *malware*. Here, we mostly just cover the basics—for more details, the place to start is Aycock's fine book [21].

Malware can be subdivided into many different categories. We'll use the following classification system, although there is considerable overlap between the various types.

- A *virus* is malware that relies on someone or something else to propagate from one system to another. For example, an email virus attaches itself to an email that is sent from one user to another. Until recently, viruses were the most popular form of malware.[8]
- A *worm* is like a virus except that it propagates by itself without the need for outside assistance. This definition implies that a worm uses a network to spread its infection.
- A *trojan horse*, or trojan, is software that appears to be one thing but has some unexpected functionality. For example, an innocent-looking game could do something malicious while the victim is playing.
- A *trapdoor* or *backdoor* allows unauthorized access to a system.
- A *rabbit* is a malicious program that exhausts system resources. Rabbits could be implemented using viruses, worms, or other means.
- *Spyware* is a type of malware that monitors keystrokes, steals data or files, or performs some similar function [22].

Generally, we won't be too concerned with placing a particular piece of malware into its precise category. We'll use the term virus as shorthand for a virus, worm, or other such malware. It is worth noting that many "viruses" (in popular usage of the term) are not viruses in the technical sense.

Where do viruses live on a system? It should come as no surprise that *boot sector* viruses live in the boot sector, where they are able to take control early in the boot process. Such a virus can then take steps to mask its presence before it can be detected. From a virus writer's perspective, the boot sector is a good place to be.

[8]The term "virus" is sometimes reserved for parasitic malware, that is, malware that relies on other code to perform its intended function.

Another class of viruses are *memory resident*, meaning that they stay in memory. Rebooting the system may be necessary to flush these viruses out. Viruses also can live in applications, macros, data, library routines, compilers, debuggers, and even in virus checking software.

By computing standards, malware is ancient. The first substantive work on viruses was done by Fred Cohen in the 1980s [62], who clearly demonstrated that malware could be used to attack computer systems.[9]

Arguably, the first virus of any significance to appear in the wild was the so-called Brain virus of 1986. Brain did nothing malicious, and it was considered little more than a curiosity. As a result, it did not awaken people to the security implications of malware. That complacency was shaken in 1988 when the Morris Worm appeared. In spite of its early date, the Morris Worm remains one of the more interesting pieces of malware to date, and we'll have more to say about it below. The other examples of malware that we'll discuss in some detail are Code Red, which appeared in 2001, and SQL Slammer, which appeared in January of 2003. We'll also present a simple example of a trojan and we'll discuss the future of malware. For more details on many aspects of malware—including good historical insights—see [66].

11.3.1 Brain

The Brain virus of 1986 was more annoying than harmful. Its importance lies in the fact that it was first, and as such it became a prototype for many later viruses. But because it was not malicious, there was little reaction by users. In retrospect, Brain provided a clear warning of the potential for malware to cause damage, but at the time that warning was mostly ignored. In any case, computing systems remained extremely vulnerable to malware.

Brain placed itself in the boot sector and other places on the system. It then screened all disk access so as to avoid detection and to maintain its infection. Each time the disk was read, Brain would check the boot sector to see if it was infected. If not, it would reinstall itself in the boot sector and elsewhere. This made it difficult to completely remove the virus. For more details on Brain, see Chapter 7 of Robert Slade's excellent history of viruses [66].

11.3.2 Morris Worm

Information security changed forever when the eponymous Morris Worm attacked the Internet in 1988 [37, 229]. It's important to realize that the Internet of 1988 was nothing like the Internet of today. Back then, the Internet was populated by academics who exchanged email and used `telnet` for remote

[9]Cohen credited Len Adleman (the "A" in RSA) with coining the term "virus."

access to supercomputers. Nevertheless, the Internet had reached a critical mass that made it vulnerable to self-sustaining worm attacks.

The Morris Worm was a cleverly designed and sophisticated piece of software that was written by a lone graduate student at Cornell University.[10] Morris claimed that his worm was a test gone bad. In fact, the most serious consequence of the worm was due to a flaw (according to Morris). In other words, the worm had a bug.

The Morris Worm was apparently supposed to check whether a system was already infected before trying to infect it. But this check was not always done, and so the worm tried to re-infect already infected systems, which led to resource exhaustion. So the (unintended) malicious effect of the Morris Worm was essentially that of a so-called rabbit.

Morris' worm was designed to do the following three things.

- Determine where it could spread its infection
- Spread its infection wherever possible
- Remain undiscovered

To spread its infection, the Morris worm had to obtain remote access to machines on the network. To gain access, the worm attempted to guess user account passwords. If that failed, it tried to exploit a buffer overflow in `fingerd` (part of the Unix `finger` utility), and it also tried to exploit a trapdoor in `sendmail`. The flaws in `fingerd` and `sendmail` were well known at the time but not often patched.

Once access had been obtained to a machine, the worm sent a bootstrap loader to the victim. This loader consisted of 99 lines of C code that the victim machine compiled and executed. The bootstrap loader then fetched the rest of the worm. In this process, the victim machine even authenticated the sender.

The Morris worm went to great lengths to remain undetected. If the transmission of the worm was interrupted, all of the code that had been transmitted was deleted. The code was also encrypted when it was downloaded, and the downloaded source code was deleted after it was decrypted and compiled. When the worm was running on a system, it periodically changed its name and process identifier (PID), so that a system administrator would be less likely to notice anything unusual.

It's no exaggeration to say that the Morris Worm shocked the Internet community of 1988. The Internet was supposed to be able to survive a nuclear attack, yet it was brought to its knees by a graduate student and a few

[10]As if to add a conspiratorial overtone to the the entire affair, Morris' father worked at the super-secret National Security Agency at the time [248].

hundred lines of C code. Few, if any, had imagined that the Internet was so vulnerable to such an attack.

The results would have been much worse if Morris had chosen to have his worm do something truly malicious. In fact, it could be argued that the greatest damage was caused by the widespread panic the worm created—many users simply pulled the plug, believing it to be the only way to protect their system. Those who stayed online were able to receive some information and therefore recovered more quickly than those who chose to rely on the infallible "air gap" firewall.

As a direct result of the Morris Worm, the Computer Emergency Response Team (CERT) [51] was established, which continues to be a primary clearinghouse for timely computer security information. While the Morris Worm did result in increased awareness of the vulnerability of the Internet, curiously, only limited actions were taken to improve security. This event should have served as a wakeup call and could well have led to a complete redesign of the security architecture of the Internet. At that point in history, such a redesign effort would have been relatively easy, whereas today it is completely infeasible. In that sense, the Morris Worm can be seen as a missed opportunity.

After the Morris Worm, viruses became the mainstay of malware writers. Only relatively recently have worms reemerged in a big way. Next, we'll consider two worms that indicate some of the trends in malware.

11.3.3 Code Red

When Code Red appeared in July of 2001, it infected more than 300,000 systems in about 14 hours. Before Code Red had run its course, it infected several hundred thousand more, out of an estimated 6,000,000 susceptible systems worldwide. To gain access to a system, the Code Red worm exploited a buffer overflow in Microsoft IIS server software. It then monitored traffic on port 80, looking for other potential targets.

The action of Code Red depended on the day of the month. From day 1 to 19, it tried to spread its infection, then from day 20 to 27 it attempted a distributed denial of service (DDoS) attack on `www.whitehouse.gov`. There were many copycat versions of Code Red, one of which included a trapdoor for remote access to infected systems. After infection, this variant flush all traces of the worm, leaving only the trapdoor.

The speed at which Code Red infected the network was something new and, as a result, it generated a tremendous amount of hype [72]. For example, it was claimed that Code Red was a "beta test for information warfare" [235]. However, there was (and still is) no evidence to support such claims or any of the other general hysteria that surrounded the worm.

11.3.4 SQL Slammer

The SQL Slammer worm burst onto the scene in January of 2003, when it infected at least 75,000 systems within 10 minutes. At its peak, the number of Slammer infections doubled every 8.5 seconds [209].

The graphs in Figure 11.16 show the increase in Internet traffic as a result of Slammer. The graph on the bottom shows the increase over a period of hours (note the initial spike), while the graph on the top shows the increase over the first five minutes.

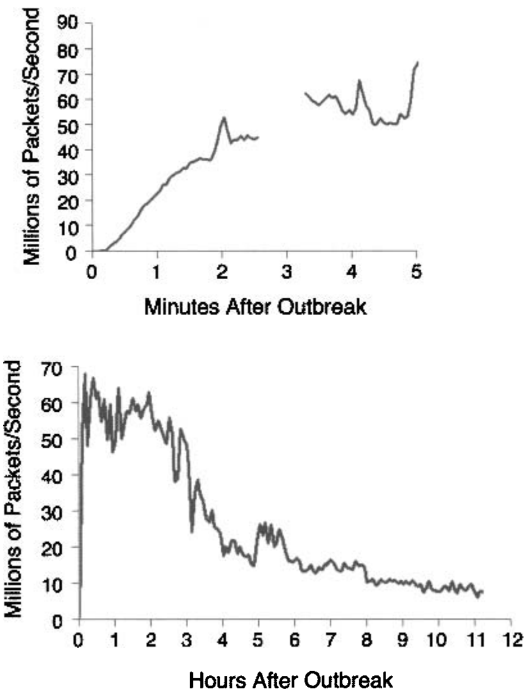


Figure 11.16: Slammer and Internet Traffic

The reason that Slammer created such a spike in Internet traffic is that each infected site searched for new susceptible sites by randomly generating IP addresses. A more efficient search strategy would have made more effective use of the available bandwidth. We'll return to this idea below when we discuss the future of malware.

It's been claimed (with good supporting evidence) that Slammer spread too fast for its own good, and effectively burned out the available bandwidth on the Internet [92]. In other words, if Slammer had been able to throttle itself slightly, it could have ultimately infected more systems and it might have caused significantly more damage.

Why was Slammer so successful? For one thing, the entire worm fit into a single 376-byte UDP packet. Firewalls are often configured to let sporadic packets through, on the theory that a single small packet can do no harm by itself. The firewall then monitors the "connection" to see whether anything unusual occurs. Since it was generally expected that much more that 376 bytes would be required for an attack, Slammer succeeded in large part by defying the assumptions of the security experts.

11.3.5 Trojan Example

In this section, we'll present a trojan, that is, a program that has some unexpected function. This trojan comes from the Macintosh world, and it's totally harmless, but its creator could just as easily have had this program do something malicious [103]. In fact, the program could have done anything that a user who executed the program could do.

This particular trojan appears to be audio data, in the form of an mp3 file that we'll name `freeMusic.mp3`. The icon for this file appears in Figure 11.17. A user would expect that double clicking on this file would automatically launch iTunes, and play the music contained in the mp3 file.

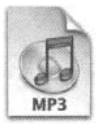
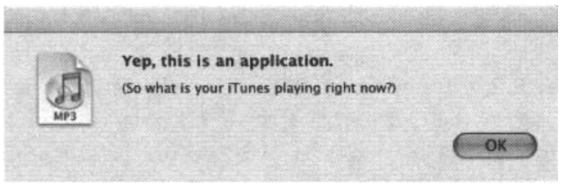


Figure 11.17: Icon for `freeMusic.mp3`

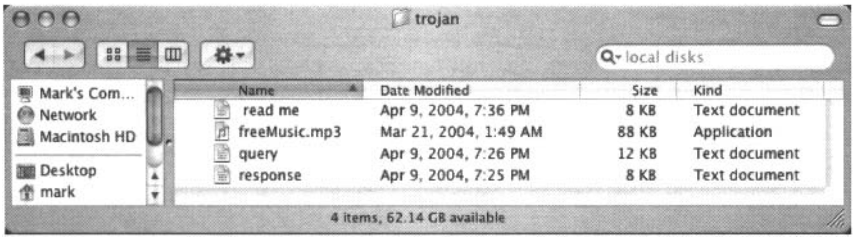
After double-clicking on the icon in Figure 11.17, iTunes launches (as expected) and an mp3 file titled "Wild Laugh" is played (probably not expected). Simultaneously, and unexpectedly, the message window in Figure 11.18 appears.


Yep, this is an application.

(So what is your iTunes playing right now?)

Figure 11.18: Unexpected Effect of `freeMusic.mp3` Trojan

What just happened? This "mp3" file is a wolf in sheep's clothing—the file `freeMusic.mp3` is not an mp3 file at all. Instead it's an application (that is, an executable file) that has had its icon changed so that it appears to be an mp3 file. A careful look at `freeMusic.mp3` reveals this fact, as shown in Figure 11.19.



Name	Date Modified	Size	Kind
read me	Apr 9, 2004, 7:36 PM	8 KB	Text document
freeMusic.mp3	Mar 21, 2004, 1:49 AM	88 KB	Application
query	Apr 9, 2004, 7:26 PM	12 KB	Text document
response	Apr 9, 2004, 7:25 PM	8 KB	Text document

Figure 11.19: Trojan Revealed

Most users are unlikely to give a second thought to opening a file that appears to be an mp3. This trojan only issues a harmless warning, but that's because the author had no malicious intent and instead simply wanted to illustrate a point [160].

11.3.6 Malware Detection

There are three general approaches that are used to detect malware. The first, and most common, is *signature detection*, which relies on finding a pattern or signature that is present in a particular piece of malware. A second approach is *change detection*, which detects files that have changed. A file that has unexpectedly changed might indicate an infection. The third approach is *anomaly detection*, where the goal is to detect unusual or virus-like files or behavior. We'll briefly discuss each of these approaches and consider their relative advantages and disadvantages.

In Chapter 8, we discussed signature-based and anomaly-based intrusion detection systems (IDSs). There are many parallels between IDSs and the corresponding virus detection methods.

11.3.6.1 Signature Detection

A signature is generally a string of bits found in a file, which might include wildcards. A hash value could also serve as a signature, but it would be less flexible and easier for virus writers to defeat.

For example, according to [296], the signature used for the W32/Beast virus is `83EB 0274 EB0E 740A 81EB 0301 0000`. We can search for this signature in all files on a system. However, if we find the signature, we can't be certain that we've found the virus, since other innocent files could contain the same string of bits. If the bits in searched files were random, the chance of such a false match would be $1/2^{112}$, which is negligible. However, computer software and data is far from random, so there is probably some realistic chance of a false match. This means that if a matching signature is found, further testing may be required to be certain that it actually represents the W32/Beast virus.

Signature detection is highly effective on malware that is known and for which a common signature can be extracted. Another advantage of signature detection is that it places a minimal burden on users and administrators, since all that is required is to keep signature files up to date and periodically scan for viruses.

A disadvantage of signature detection is that signature files can become large—tens or hundreds of thousands of signatures is the norm—which can make scanning slow. Also, the signature files must be kept up to date. A more fundamental problem is that we can only detect known signatures. Even a slight variant of a known virus might be missed.

Today, signature detection is by far the most popular malware detection method. As a result, virus writers have developed some sophisticated means for avoiding signature detection. We'll have more to say about this below.

11.3.6.2 Change Detection

Since malware must reside somewhere, if we detect a change somewhere on a system, then it may indicate an infection. That is, if we detect that a file has changed, it may be infected with a virus. We'll refer to this approach as change detection.

How can we detect changes? Hash functions are useful in this regard. Suppose we compute hashes of all files on a system and securely store these hash values. Then at regular intervals we can recompute the hashes and compare the new values with the stored values. If a file has changed in one or more bits—as it will in the case of a virus infection—we'll find that the computed hash does not match the previously computed hash value.

One advantage of change detection is that there are virtually no false negatives, that is, if a file has been infected, we'll detect a change. Another major advantage is that we can detect previously unknown malware (a change is a change, whether it's caused by a known or unknown virus).

However, the disadvantages to change detection are many. Files on a system often change and as a result there will be many false positives, which places a heavy burden on users and administrators. If a virus is inserted into a file that changes often, it will be more likely to slip through a change detection regimen. And what should be done when a suspicious change is detected? A careful analysis of log files might prove useful. But, in the end, it might be necessary to fall back to a signature scan, in which case the advantages of change detection have been largely negated.

11.3.6.3 Anomaly Detection

Anomaly detection is aimed at finding any unusual or virus-like or other potentially malicious activity or behavior. We discussed this idea in detail Chapter 8 when we covered intrusion detection systems (IDSs), so we only briefly discuss the concepts here.

The fundamental challenge with anomaly detection lies in determining what is normal and what is unusual, and being able to distinguish between the two. Another serious difficulty is that the definition of normal can change, and the system must adapt to such changes, or it will likely overwhelm users with false alarms.

The major advantage of anomaly detection is that there is some hope of detecting previously unknown malware. But, as with change detection, the disadvantages are many. For one, anomaly detection is largely unproven in practice. Also, as discussed in the IDS section of Chapter 8, a patient attacker may be able to make an anomaly appear to be normal. In addition, anomaly detection is not robust enough to be used as a standalone detection system, so it is usually combined with a signature detection system.

In any case, many people have very high hopes for the ultimate success of anomaly detection. However, today anomaly detection is primarily a challenging research problem rather than a practical security solution.

Next, we'll discuss some aspects of the future of malware. This discussion should make it clear that better malware detection tools will be needed, and sooner rather than later.

11.3.7 The Future of Malware

What does the future hold for malware? Below, we'll briefly consider a few possible attacks. Given the resourcefulness of malware developers, we can expect to see attacks based on these or similar ideas in the future [24, 289].

But before we discuss the future, let's briefly consider the past. Virus writers and virus detectors have been locked in mortal combat since the first virus detection software appeared. For each advance in detection, virus writers have responded with strategies that make their handiwork harder to detect.

One of the first responses of virus writers to the success of signature detection systems was *encrypted* malware. If an encrypted worm uses a different key each time it propagates, there will be no common signature. Often the encryption is extremely weak, such as a repeated XOR with a fixed bit pattern. The purpose of the encryption is not confidentiality, but to simply mask any possible signature.

The Achilles heel of encrypted malware is that it must include decryption code, and this code is subject to signature detection. The decryption routine typically includes very little code, making it more difficult to obtain a signature, and yielding more cases requiring secondary testing. The net result is that signature scanning can be applied, but it will be slower than for unencrypted malware.

The next step in the evolution of malware was the use of *polymorphic* code. In a polymorphic virus the body is encrypted and the decryption code is morphed. Consequently, the signature of the virus itself (i.e., the body) is hidden by encryption, while the decryption code has no common signature due to the morphing.

Polymorphic malware can be detected using emulation. That is, suspicious code can be executed in an emulator. If the code is malware, it must eventually decrypt itself, at which point standard signature detection can be applied to the body. This type of detection will be much slower than a simple signature scan due to the emulation.

Metamorphic malware takes polymorphism to the limit. A metamorphic worm mutates before infecting a new system.[11] If the mutation is sufficient, such a worm can likely avoid any signature-based detection system. Note that the mutated worm must do the same thing as the original worm, but yet its internal structure must be different enough to avoid detection. Detection of metamorphic software is currently a challenging research problem [297].

Let's consider how a metamorphic worm might replicate [79]. First, the worm could disassemble itself and then strip the resulting code to a base form. Randomly selected blocks of code could be inserted into the assembly. These variations could include, for example, rearranging jumps and inserting dead code. The resulting code would then be assembled to obtain a worm with the same functionality as the original, but it would be unlikely to have a common signature.

While the metamorphic generator described in the previous paragraph sounds plausible, in reality it is surprisingly difficult to produce highly metamorphic code. As of the time of this writing, the hacker community has produce a grand total of one reasonably metamorphic generator. These and related topics are discussed in the series of papers [193, 279, 312, 330].

Another distinct approach that virus writers have pursued is speed. That is, viruses such as Code Red and Slammer have tried to infect as many machines as possible in as short of a time as possible. This can also be viewed as an attack aimed at defeating signature detection, since a rapid attack would not allow time for signatures to be extracted and distributed.

According to the late pop artist Andy Warhol, "In the future everybody will be world-famous for 15 minutes" [301]. A *Warhol worm* is designed to infect the entire Internet in 15 minutes or less. Recall that Slammer infected a large number of systems in 10 minutes. Slammer burned out the available

[11]Metamorphic malware is sometimes called "body polymorphic," since polymorphism is applied to the entire virus body.

bandwidth due to the way that it searched for susceptible hosts, and as a result, Slammer was too bandwidth-intensive to have infected the entire Internet in 15 minutes. A true Warhol worm must do "better" than Slammer. How is this possible?

One plausible approach is the following. The malware developer would do preliminary work to develop an initial "hit list" of sites that are susceptible to the particular exploit used by the worm. Then the worm would be seeded with this hit list of vulnerable IP addresses. Many sophisticated tools exist for identifying systems and these could help to pinpoint systems that are susceptible to a given attack.

When this Warhol worm is launched, each of the sites on its initial hit list will be infected since they are all known to be vulnerable. Then each of these infected sites can scan a predetermined part of IP address space looking for additional victims. This approach would avoid duplication and the resulting wasted bandwidth that caused Slammer to bog down.

Depending on the size of the initial hit list, the approach described above could conceivably infect the entire Internet in 15 minutes or less. No worm this sophisticated has yet been seen in the wild. Even Slammer relied on randomly generated IP addresses to spread its infection.

Is it possible to do "better" than a Warhol worm? That is, can the entire Internet be infected in significantly less than 15 minutes? A *flash worm* is designed to infect the entire Internet almost instantly.

Searching for vulnerable IP addresses is the slow part of any worm attack. The Warhol worm described above uses a smarter search strategy, where it relies on an initial list of susceptible systems. A flash worm could take this approach to the limit by embedding all susceptible IP addresses into the worm.

A great deal of work would be required to predetermine all vulnerable IP addresses, but there are hacker tools available that would significantly reduce the burden. Once all vulnerable IP addresses are known, the list could be partitioned between several initial worm variants. This would still result in large worms [79], but each time the worm replicates, it would split the list of addresses embedded within it, as illustrated in Figure 11.20. Within a few generations the worm would be reduced to a reasonable size. The strength of this approach is that it results in virtually no wasted time or bandwidth.

It has been estimated that a well-designed flash worm could infect the entire Internet in as little as 15 seconds! Since this is much faster than humans could possibly respond, any defense against such an attack must be automated. A conjectured defense against flash worms [79] would be to deploy many personal intrusion detection systems and to have a master IDS monitor these personal IDSs. When the master IDS detects unusual activity, it can let it proceed on a few nodes, while temporarily blocking it elsewhere. If the sacrificial nodes are adversely affected, then an attack is in progress,

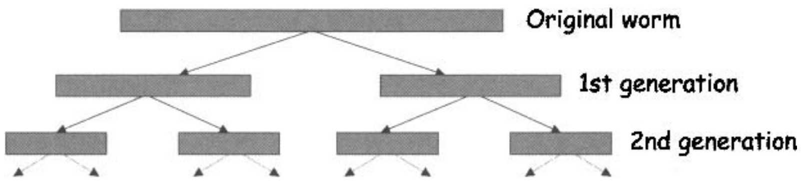


Figure 11.20: A Flash Worm

and it can be blocked elsewhere. On the other hand, if it's a false alarm, the other nodes are only delayed slightly. This defensive strategy shares many of the challenges associated with anomaly-based intrusion detection systems, as discussed in Chapter 8.

11.3.8 Cyber Diseases Versus Biological Diseases

It's currently fashionable to make biological analogies with computing. There are many such analogies that are applied to the field of security. In the field of malware and in particular, computer viruses, the analogy is fairly obvious.

There clearly are similarities between biological and computer "diseases." For example, in nature, if there are too few susceptible individuals, a disease will die out. A somewhat similar situation exists on the Internet, where too few susceptible systems may not allow a worm to become self-sustaining, particularly if the worm is randomly searching for vulnerable IP addresses.

There are, however, some significant differences between cyber diseases and biological diseases. For example, there is virtually no sense of distance on the Internet, so many of the models developed for biological diseases don't apply to cyber diseases.[12] Also, in nature, diseases attack more or less at random, while in computer systems hackers often specifically target the most desirable or vulnerable systems. As a result, computer attacks are potentially more focused and damaging than biological diseases. The important point here is that, although the biological analogy is useful, it cannot be taken too literally.

Finally, we note in passing that cell phones have not been plagued with malware to nearly the same degree as computer systems. Various explanations for this phenomenon have been given, with two of the more plausible being the relative diversity of mobile systems and inherently stronger security architectures. For a discussion of the Android security architecture and some of the difficulties of mounting a successful attack, see [211].

[12]However, with some cell phone attacks, proximity is required (e.g., attacks that rely on Bluetooth) while network-based attacks are also possible. So, cell phone attacks could include aspects of both biological viruses and computer viruses.

11.4 Botnets

A *botnet* is a collection of a large number of compromised machines under the control of a *botmaster*. The name derives from the fact that individual compromised machines are known as *bots* (shorthand for robots). In the past, such machines were often known as zombies.

Until recently, botmasters typically employed the Internet Relay Chat (IRC) protocol to manage their bots. However, newer botnets often use Peer-to-Peer (P2P) architectures since these are more difficult for authorities to track and shut down.

Botnets have proven ideal tools for sending spam and for launching distributed denial of service (DDoS) attacks. For example, a botnet was used in a highly-publicized denial of service attack on Twitter that was apparently aimed at silencing one well-known blogger from the Republic of Georgia [207].[13]

Botnets are a hot security topic, but at this point in time their activities in the wild are not completely understood. For example, there are wildly differing estimates for the sizes of various botnets [224].

Finally, it is often claimed that in the past most attacks were conducted primarily for fame within the hacker community, or for ideological reasons, or by script kiddies with little knowledge of what they were actually doing. That is, attacks were essentially just malicious pranks. In contrast (or so the claim goes), today attacks are primarily for profit. Some even believe that organized crime is behind most current attacks. The profit motive is plausible since earlier widespread attacks (Code Red, Slammer, etc.) were first and foremost designed to make headlines, whereas botnets strive to remain undetected. In addition, botnets are ideal for use in various subtle attack-for-hire scenarios. Of course, you should always be skeptical of those who hype any supposed threat, especially when they have a vested interest in the hype becoming conventional wisdom.[14]

11.5 Miscellaneous Software-Based Attacks

In this section we'll consider a few software-based attacks that don't fit neatly into any of our previous discussion. While there are numerous such attacks, we'll restrict our attention to a few representative examples. The topics we'll discuss are *salami attacks*, *linearization attacks*, *time bombs*, and the general issue of trusting software.

[13]Of course, this raised suspicion that Russian government intelligence agencies were behind the attack. However, the attack accomplished little, other than greatly increasing the fame of the attackee, so it's difficult to believe that any intelligence agency would be so stupid. On the other hand, "government intelligence" is an oxymoron.

[14]Or, more succinctly, "Beware the prophet seeking profit" [205].

11.5.1 Salami Attacks

In a salami attack, a programmer slices off a small amount of money from individual transactions, analogous to the way that you might slice off thin pieces from a salami.[15] These slices must be difficult for the victim to detect. For example, it's a matter of computing folklore that a programmer at a bank can use a salami attack to slice off fractional cents leftover from interest calculations. These fractional cents—which are not noticed by the customers or the bank—are deposited in the programmer's account. Over time, such an attack could prove highly lucrative for the dishonest programmer.

There are many confirmed cases of salami attacks. The following examples all appear in [158]. In one documented case, a programmer added a few cents to every employee payroll tax withholding calculation, but credited the extra money to his own tax. As a result, this programmer got a hefty tax refund. In another example, a rent-a-car franchise in Florida inflated gas tank capacity so it could overcharge customers for gas. An employee at a Taco Bell location reprogrammed the cash register for the late-night drive-through line so that \$2.99 specials registered as \$0.01. The employee then pocketed the \$2.98 difference—a rather large slice of salami!

In a particularly clever salami attack, four men who owned a gas station in Los Angeles hacked a computer chip so that it overstated the amount of gas pumped. Not surprisingly, customers complained when they had to pay for more gas than their tanks could hold. But this scam was hard to detect, since the gas station owners were clever. They had programmed the chip to give the correct amount of gas whenever exactly 5 or 10 gallons was purchased, because they knew from experience that inspectors usually ask for 5 or 10 gallons. It took multiple inspections before they were caught.

11.5.2 Linearization Attacks

Linearization is an approach that is applicable in a wide range of attacks, from traditional lock picking to state-of-the-art cryptanalysis. Here, we consider an example related to breaking software, but it is important to realize that this concept has wide application.

Consider the program in Table 11.5, which checks an entered number to determine whether it matches the correct serial number. In this case, the correct serial number happens to be S123N456. For efficiency, the programmer decided to check one character at a time and to quit checking as soon as one incorrect character is found. From a programmer's perspective, this is a perfectly reasonable way to check the serial number, but it might open the door to an attack.

[15]Or the name might derive from the fact that a salami consists of bunch of small undesirable pieces that are combined to yield something of value.

Table 11.5: Serial Number Program

```
int main(int argc, const char *argv[])
{
   int i;
   char serial[9]="S123N456\n";
   if(strlen(argv[1]) < 8)
   {
      printf("\nError---try again.\n\n");
      exit(0);
   }
   for(i = 0; i < 8; ++i)
   {
      if(argv[1][i] != serial[i]) break;
   }
   if(i == 8)
   {
      printf("\nSerial number is correct!\n\n");
   }
}
```

How can Trudy take advantage the code in Table 11.5? Note that the correct serial number will take longer to process than any incorrect serial number. More precisely, the more leading characters that are correct, the longer the program will take to check the number. So, a putative serial number that has the first character correct will take longer than any that has an incorrect first character. Therefore, Trudy can select an eight-character string and vary the first character over all possibilities. If she can time the program precisely enough, she will find that the string beginning with S takes the most time. Trudy can then fix the first character as S and vary the second character, in which case she will find that a second character of 1 takes the longest. Continuing, Trudy can recover the serial number one character at a time. That is, Trudy can attack the serial number in linear time, instead of searching an exponential number of cases.

How great is the advantage for Trudy in this linearization attack? Suppose the serial number is eight characters long and each character has 128 possible values. Then there are $128^8 = 2^{56}$ possible serial numbers. If Trudy must randomly guess complete serial numbers, she would obtain the serial number in about 2^{55} tries, which is an enormous amount of work. On the other hand, if she can use a linearization attack, an average of only $128/2 = 64$ guesses are required for each letter, for a total expected work of about $8 \cdot 64 = 2^9$. This makes an otherwise infeasible attack into a trivial attack.

A real-world example of a linearization attack occurred in TENEX [235], a timeshare system used in ancient times.[16] In TENEX, passwords were verified one character at a time, so the system was subject to a linearization attack similar to the one described above. However, careful timing was not even necessary. Instead, it was possible to arrange for a "page fault" to occur when the next unknown character was guessed correctly. Then a user-accessible page fault register would tell the attacker that a page fault had occurred and, therefore, that the next character had been guessed correctly. This attack could be used to crack any password in seconds.

11.5.3 Time Bombs

Time bombs are another interesting class of software-based attacks. We'll illustrate the concept with an infamous example. In 1986, Donald Gene Burleson told his employer to stop withholding taxes from his paycheck. Since this isn't legal, the company refused. Burleson, a tax protester, made it known that he planned to sue his company. Burleson used company time and resources to prepare his legal case against his company. When the company discovered what Burleson was doing, they fired him [240].

It later came to light that Burleson had been developing malicious software. After he was fired, Burleson triggered his "time bomb" software, which proceeded to delete thousands of records from the company's computer.

The Burleson story doesn't end here. Out of fear of embarrassment, the company was reluctant to pursue a legal case, despite their losses. Then in a bizarre twist, Burleson sued his former employer for back pay, at which point the company finally sued Burleson. The company eventually won, and in 1988 Burleson was fined $11,800. The case took two years to prosecute at a cost of tens of thousands of dollars and resulted in little more than a slap on the wrist. The light sentence was likely due to the fact that laws regarding computer crime were unclear at that early date. In any case, this was one of the first computer crime cases in the United States, and many cases since have followed a similar pattern. In particular, companies are often reluctant to pursue such cases for fear that it will damage their reputation.

11.5.4 Trusting Software

Finally, we consider a philosophical question with practical significance: Can you ever trust software? In the fascinating article [303], the following thought experiment is discussed. Suppose that a C compiler has a virus. When

[16]The 1960s and 1970s, that is. In computing, that's the age when dinosaurs roamed the earth.

compiling the `login` program, this virus creates a backdoor in the form of an account with a known password. Also, if the C compiler is recompiled, the virus incorporates itself into the newly compiled C compiler.

Now suppose that you suspect that your system is infected with a virus. You want to be absolutely certain that you fix the problem, so you decide to start over from scratch. You recompile the C compiler, then use it to recompile the operating system, which includes the `login` program. You haven't gotten rid of the problem, since the backdoor was once again compiled into the `login` program.

Analogous situations could arise in the real world. For example, imagine that an attacker is able to hide a virus in your virus scanning software. Or consider the damage that could be done by a successful attack on online virus signature updates—or other automated software updates.

Software-based attacks might not be obvious, even to an expert who examines the source code line by line. For example, in the Underhanded C Contest, the rules state in part that [70]

> ...in this contest you must write code that is as readable, clear, innocent and straightforward as possible, and yet it must fail to perform at its apparent function. To be more specific, it should do something subtly evil.

Some of the programs submitted to this contest are extremely subtle and they demonstrate that it is possible to make evil code look innocent.

We'll return to the theme of trusting software when we discuss operating systems in Chapter 13. Specifically, we will outline an ambitious design for a trusted operating system.

11.6 Summary

In this chapter, we discussed some of the security threats that arise from software. The threats considered here come in two basic flavors. The plain vanilla flavor consists of unintentional software flaws that attackers can sometimes exploit. The classic example of such a flaw is the buffer overflow, which we discussed in some detail. Another common flaw with security implications is a race condition.

The more exotic flavor of software security threats arise from intentionally malicious software, or malware. Such malware includes the viruses and worms that plague users today, as well as trojans and backdoors. Malware writers have developed highly sophisticated techniques for avoiding detection, and they appear set to push the envelope much further in the near future. Whether detection tools are up to the challenge posed by the next generation of malware is an open question.

11.7 Problems

1. With respect to security, it's been said that complexity, extensibility, and connectivity are the "trinity of trouble" [143]. Define each of these terms and explain why each represents a potential security problem.

2. What is a validation error, and how can such an error lead to a security flaw?

3. Provide a detailed discussion of one real-world virus or worm that was not covered in the text.

4. What is a race condition? Discuss an example of a real-world race condition, other than the `mkdir` example presented in the text.

5. One type of race condition is known as a time-of-check-to-time-of-use, or TOCTTOU (pronounced "TOCK too").

    a. What is a TOCTTOU race condition and why is it a security issue?
    b. Is the `mkdir` race condition discussed in this chapter an example of a TOCTTOU race condition?
    c. Give two real-world examples of TOCTTOU race conditions.

6. Recall that a canary is a special value that is pushed onto the stack after the return address.

    a. How is a canary used to prevent stack smashing attacks?
    b. How was Microsoft's implementation of this technique, the `/GS` compiler option, flawed?

7. Discuss one real-world example of a buffer overflow that was exploited as part of a successful attack.

8. Explain how a heap-based buffer overflow works, in contrast to the stack-based buffer overflow discussed in this chapter.

9. Explain how an integer overflow works, in contrast to the stack-based buffer overflow discussed in this chapter.

10. Read the article [311] and explain why the author views the NX bit as only one small part of the solution to the security problems that plague computers today.

11. As discussed in the text, the C function `strcpy` is unsafe. The C function `strncpy` is a safer version of `strcpy`. Why is `strncpy` safer but not safe?

12. Suppose that Alice's system employs the NX bit method of protecting against buffer overflow attacks. If Alice's system uses software that is known to harbor multiple buffer overflows, would it be possible for Trudy to conduct a denial of service attack against Alice by exploiting one of these buffer overflows? Explain.

13. Suppose that the NX bit method of protecting against buffer overflow attacks is employed.

    a. Will the buffer overflow illustrated in Figure 11.5 succeed?

    b. Will the attack in Figure 11.6 succeed?

    c. Why will the return-to-libc buffer overflow example discussed in Section 11.2.1.2 succeed?

14. List all unsafe C functions and explain why each is unsafe. List the safer alternative to each and explain whether each is safe or only safer, as compared to its unsafe alternative.

15. In addition to stack-based buffer overflow attacks (i.e., smashing the stack), heap overflows can also be exploited. Consider the following C code, which illustrates a heap overflow.

```
int main()
{
    int diff, size = 8;
    char *buf1, *buf2;
    buf1 = (char *)malloc(size);
    buf2 = (char *)malloc(size);
    diff = buf2 - buf1;
    memset(buf2, '2', size);
    printf("BEFORE: buf2 = %s ", buf2);
    memset(buf1, '1', diff + 3);
    printf("AFTER: buf2 = %s ", buf2);
    return 0;
}
```

    a. Compile and execute this program. What is printed?

    b. Explain the results you obtained in part a.

    c. Explain how a heap overflow might be exploited by Trudy.

16. In addition to stack-based buffer overflow attacks (i.e., smashing the stack), integer overflows can also be exploited. Consider the following C code, which illustrates an integer overflow [36].

```
int copy_something(char *buf, int len)
{
   char kbuf[800];
   if(len > sizeof(kbuf))
   {
      return -1;
   }
   return memcpy(kbuf, buf, len);
}
```

a. What is the potential problem with this code? Hint: The last argument to the function `memcpy` is interpreted as an unsigned integer.

b. Explain how an integer overflow might be exploited by Trudy.

17. Obtain the file `overflow.zip` from the textbook website and extract the Windows executable.

    a. Exploit the buffer overflow so that you bypass its serial number check. Turn in a screen capture to verify your success.

    b. Determine the correct serial number.

18. Consider the following protocol for adding money to a debit card.

    (i) User inserts debit card into debit card machine.

    (ii) Debit card machine determines current value of card (in dollars), which is stored in variable x.

    (iii) User inserts dollars into debit card machine and the value of the inserted dollars is stored in variable y.

    (iv) User presses `enter` button on debit card machine.

    (v) Debit card machine writes value of $x + y$ dollars to debit card and ejects card.

    Recall the discussion of race conditions in the text. This particular protocol has a race condition.

    a. What is the race condition in this protocol?

    b. Describe a possible attack that exploits the race condition.

    c. How could you change the protocol to eliminate the race condition, or at least make it more difficult to exploit?

19. Recall that a trojan horse is a program that has unexpected functionality.

  a. Write your own trojan horse, where the unexpected functionality is completely harmless.

  b. How could your trojan program be modified to do something malicious?

20. Recall that a computer virus is malware that relies on someone or something (other than itself) to propagate from one system to another.

    a. Write your own computer virus, where the "malicious" activity is completely harmless.

    b. Explain how your virus could be modified to do something malicious.

21. Recall that a worm is a type of malware similar to a virus except that a worm propagates by itself.

    a. Write your own worm, where the "malicious" activity is completely harmless.

    b. Explain how your worm could be modified to do something malicious.

22. Virus writers use encryption, polymorphism, and metamorphism to evade signature detection.

    a. What are the significant differences between encrypted worms and polymorphic worms?

    b. What are the significant differences between polymorphic worms and metamorphic worms?

23. This problem deals with metamorphic software.

    a. Define metamorphic software.

    b. Why would a virus writer employ metamorphic techniques?

    c. How might metamorphic software be used for good instead of evil?

24. Suppose that you are asked to design a metamorphic generator. Any assembly language program can be given as input to your generator, and the output must be a metamorphic version of the input program. That is, your generator must produce a morphed version of the input program and this morphed code must be functionally equivalent to the input program. Furthermore, each time your generator is applied to the same input program, it must, with high probability, produce a distinct metamorphic copy. Finally, the more variation in the metamorphic copies, the better. Outline a plausible design for such a metamorphic generator.

25. Suppose that you are asked to design a metamorphic worm, where each time the worm propagates, it must first produce a morphed version of itself. Furthermore, all morphed versions must, with high probability, be distinct, and the more variation within the metamorphic copies, the better. Outline a plausible design for such a metamorphic worm.

26. A metamorphic worm that generates its own morphed copies is sometimes said to "carry its own metamorphic engine" (see Problem 25). In some situations it might be possible to instead use a standalone metamorphic generator (see Problem 24) to produce the metamorphic copies, in which case the worm would not need to carry its own metamorphic engine.

    a. Which of these two types of metamorphic worms would be easier to implement and why?
    b. Which of these two types of metamorphic worms would likely be easier to detect and why?

27. A polymorphic worm uses code morphing techniques to obfuscate its decryption code while a metamorphic worm uses code morphing techniques to obfuscate the entire worm. Apart than the amount of code that must be morphed, why is it more difficult to develop a metamorphic worm than a polymorphic worm? Assume that in either case the worm must carry its own morphing engine (see Problems 25 and 26).

28. In the paper [330] several metamorphic malware generators are tested. Curiously, all but one of the generators fail to produce any significant degree of metamorphism. Viruses from each of these weak metamorphic generators are easily detected using standard signature detection techniques. However, one metamorphic generator, known as NGVCK, is shown to produce highly metamorphic viruses, and these successfully evade signature detection by commercial virus scanners. Finally, the authors show that, in spite of the high degree of metamorphism, NGVCK viruses are relatively easy to detect using machine learning techniques—specifically, hidden Markov models [278].

    a. These results tend to indicate that the hacker community has, with rare exception, failed to produce highly metamorphic malware. Why do you suppose this is the case?
    b. It might seem somewhat surprising that the highly metamorphic NGVCK viruses can be detected. Provide a plausible explanation as to why these viruses can be detected.
    c. Is it possible to produce undetectable metamorphic viruses? If so, how? If not, why not?

29. In contrast to a flash worm, a slow worm is designed to slowly spread its infection while remaining undetected. Then, at a preset time, all of the slow worms could emerge and do something malicious. The net effect would be similar to that of a flash worm.

    a. Discuss one weakness (from Trudy's perspective) of a slow worm as compared with a flash worm.

    b. Discuss one weakness (also from Trudy's perspective) of a flash worm compared with a slow worm.

30. It has been suggested that from the perspective of signature detection, malware now far outnumbers goodware. That is, the number of signatures required to detect malicious programs exceeds the number of legitimate programs.

    a. Is it plausible that there could be more malware than legitimate programs? Why or why not?

    b. Assuming there is more malware than goodware, design an improved signature-based detection system.

31. Provide a brief discussion of each of the following botnets. Include a description of the command and control architecture and provide reasonable estimates for the maximum size and current size of each.

    a. Mariposa

    b. Conficker

    c. Kraken

    d. Srizbi

32. Phatbot, Agobot, and XtremBot all belong to the same botnet family.

    a. Pick one of these variants and discuss its command and control structure.

    b. These botnets are open source projects that are distributed under the GNU General Public License (GPL). This is highly unusual for malware—most malware writers are arrested and jailed if they are caught. Why do you suppose that the authors of these botnets are not punished?

33. In this chapter, the claim is made that "botnets are ideal for use in various attack-for-hire scenarios." Spam and various DoS attacks are the usual examples given for the uses of botnets. Give examples of other types of attacks (other than spam and DoS, that is) for which botnets would be useful.

34. After infecting a system, some viruses take steps to cleanse the system of any (other) malware. That is, they remove any malware that has previously infected the system, apply security patches, update signature files, etc.

    a. Why would it be in a virus writer's interest to protect a system from other malware?

    b. Discuss some possible defenses against malware that includes such anti-malware provisions.

35. Consider the code that appears in Table 11.5.

    a. Provide pseudo-code for a linearization attack on the code in Table 11.5.

    b. What is the source of the problem with this code, that is, why is the code susceptible to attack?

36. Consider the code in Table 11.5, which is susceptible to a linearization attack. Suppose that we modify the program as follows:

```
int main(int argc, const char *argv[])
{
   int i;
   boolean flag = true;
   char serial[9]="S123N456\n";
   if(strlen(argv[1]) < 8)
   {
      printf("\nError---try again.\n\n");
      exit(0);
   }
   for(i = 0; i < 8; ++i)
   {
      if(argv[1][i] != serial[i]) flag = false;
   }
   if(flag)
   {
      printf("\nSerial number is correct!\n\n");
   }
}
```

Note that we never break out of the `for` loop early, yet we can still determine whether the correct serial number was entered. Explain why this modified version of the program is still susceptible to a linearization attack.

37. Consider the code in Table 11.5, which is susceptible to a linearization attack. Suppose that we modify the program so that it computes the hash of the putative serial number and we compare this hash to the hash of the actual serial number. Is this modified program susceptible to a linearization attack? Explain.

38. Consider the code in Problem 36, which is susceptible to a linearization attack. Suppose that we modify the program so that it computes a random delay within each iteration of the loop.

    a. This program is still susceptible to a linearization attack. Why?

    b. An attack on this modified program would be more difficult than an attack on the code that appears in Problem 36. Why?

39. Consider the code in Table 11.5, which is susceptible to a linearization attack. Suppose that we modify the program as follows:

```
int main(int argc, const char *argv[])
{
    int i;
    char serial[9]="S123N456\n";
    if(strcmp(argv[1], serial) == 0)
    {
        printf("\nSerial number is correct!\n\n");
    }
}
```

    Note that we are using the library function `strcmp` to compare the input string to the actual serial number.

    a. Is this version of the program immune to a linearization attack? Why or why not?

    b. How is `strcmp` implemented? That is, how does it determine whether the two strings are identical or not?

40. Obtain the Windows executable contained in `linear.zip` (available at the textbook website).

    a. Use a linearization attack to determine the correct eight-digit serial number.

    b. How many guesses did you need to find the serial number?

    c. What is the expected number of guesses that would have been required if the code was not vulnerable to a linearization attack?

41. Suppose that a bank does 1000 currency exchange transactions per day.

a. Describe a salami attack on such transactions.

b. How much money would Trudy expect to make using this salami attack in a day? In a week? In a year?

c. How might Trudy get caught?

42. Consider the code in Table 11.5, which is susceptible to a linearization attack. Suppose that we modify the program as follows:

```
int main(int argc, const char *argv[])
{
   int i;
   int count = 0;
   char serial[9]="S123N456\n";
   if(strlen(argv[1]) < 8)
   {
      printf("\nError---try again.\n\n");
      exit(0);
   }
   for(i = 0; i < 8; ++i)
   {
      if(argv[1][i] != serial[i])
         count = count + 0;
      else
         count = count + 1;
   }
   if(count == 8)
   {
      printf("\nSerial number is correct!\n\n");
   }
}
```

Note that we never break out of the `for` loop early, yet we can still determine whether the correct serial number was entered. Is this version of the program immune to a linearization attack? Explain.

43. Modify the code in Table 11.5 so that it is immune to a linearization attack. Note that the resulting program must take exactly the same amount of time to execute for any incorrect input. Hint: Do not use any predefined functions (such as `strcmp` or `strncmp`) to compare the input with the correct serial number.

44. Read the article "Reflections on Trusting Trust" [303] and summarize the author's main points.