Information Security: Principles and Practice, Second Edition
by Mark Stamp

Chapter 12

Insecurity in Software

Every time I write about the impossibility of effectively protecting digital files on a general-purpose computer, I get responses from people decrying the death of copyright. "How will authors and artists get paid for their work?" they ask me. Truth be told, I don't know. I feel rather like the physicist who just explained relativity to a group of would-be interstellar travelers, only to be asked: "How do you expect us to get to the stars, then?" I'm sorry, but I don't know that, either.
— Bruce Schneier

So much time and so little to do! Strike that. Reverse it. Thank you.
— Willy Wonka

12.1 Introduction

In this chapter, we begin with software reverse engineering, or SRE. To fully appreciate the inherent difficulty of implementing security in software, we must look at software the way that attackers do. Serious attackers use SRE techniques to find and exploit flaws—or create new flaws—in software.

After our brief look at SRE, we'll discuss digital rights management, or DRM, which provides a good example of the limitation of relying on software for security. DRM illustrates the impact of SRE on software-based security.

The last major topic of this chapter is software development. It was tempting to label this section "secure software development," but truly secure software is difficult to achieve in practice. We'll discuss methods to improve the security of software, but we'll also see why most of the advantages lie with the bad guys. Finally, we briefly consider the relative security merits of open source versus closed source software.

12.2 Software Reverse Engineering

SRE or software reverse engineering—which is also known as reverse code engineering or, simply, reversing—can be used for good or for not so good. The good uses include understanding malware [336, 337] or legacy code [57]. Here, we're primarily interested in the not-so-good uses, which include removing usage restrictions from software, finding and exploiting software flaws, cheating at games, breaking DRM systems, and many, many other attacks on software.

We'll assume that the reverse engineer is our old friend Trudy. For the most part, we assume that Trudy only has an executable, or `exe`, that was generated by compiling, say, a C program. That is, Trudy does not have access to the source code. We will consider one Java reversing example, but unless obfuscation techniques have been applied, Java class files are trivial to reverse to obtain (nearly) the original source code. And even using obfuscation may not make Java significantly more difficult to reverse. On the other hand, "native code" (i.e., hardware-specific machine code) is inherently more difficult to reverse. For one thing, the best we can realistically do is disassemble an `exe` and, consequently, Trudy must analyze the program as assembly code, not as a higher-level language.

Of course, Trudy's ultimate goal is to break things. So, Trudy might reverse the software as a step toward finding a weakness or otherwise devising an attack. Often, however, Trudy wants to modify the software to bypass some annoying security feature. Before Trudy can modify the software, SRE is a necessary first step.

SRE is usually focused on software that runs under Microsoft Windows. Consequently, much of our discussion here is Windows-specific.

Essential reverse engineering tools include a *disassembler* and a *debugger*. A disassembler converts an executable into assembly code, as best it can, but a disassembler can't always disassemble code correctly, since, for example, it's not always possible to distinguish code from data. This implies that in general, it's not possible to disassemble an `exe` file and reassemble the result into a functioning executable. This will make Trudy's task slightly more challenging but by no means insurmountable.

A debugger is used to set break points, which allows Trudy to step through the code as it executes. For any reasonably complex program, a debugger is a necessary tool for understanding the code.

OllyDbg [225] includes a highly regarded debugger, disassembler, and hex editor [173]. OllyDbg is more than sufficient for all of the problems that appear in this chapter and, best of all, it's free. IDA Pro is a powerful disassembler and debugger [147]. IDA Pro costs a few hundred dollars (there is a free trial version) and it is generally considered to have the best disassembler available. Hackman [299] is an inexpensive shareware disassembler and debugger that might also be worth considering.

A hex editor can be used to directly modify, or *patch*,[1] an exe file. Today, all self-respecting debuggers include a built-in hex editor, so you may not need a standalone hex editor. But, if you should need a separate hex editor, UltraEdit and HIEW are among the most popular shareware choices.

Several other more specialized tools are sometimes useful for reverse engineering. Examples of such tools include Regmon, which monitors all accesses of the Windows registry, and Filemon, which, as you might have guessed, monitors all accesses of files. Both of these tools are available from Microsoft as freeware. VMWare [318]—which allows a user to set up virtual machines—is a powerful tool that is particularly useful if you want to reverse engineer malware while minimizing the risk of damaging your system.

Does Trudy really need a disassembler and a debugger? Note that the disassembler gives Trudy a static view of the code, which can be used to obtain an overview of the program logic. After perusing the disassembled code, Trudy can zero in on areas that are likely to be of interest. But without a debugger, Trudy would have a difficult time skipping over the boring parts of the code. Trudy would, in effect, be forced to mentally execute the code so that she could know the state of registers, variable values, flag bits, etc., at some particular point in the code. Trudy may be clever, but this would be an insurmountable obstacle for all but the simplest program.

As all software developers know, a debugger allows Trudy to set break points. In this way, Trudy can treat uninteresting parts of the code as a black box and skip directly to the interesting parts. Also, as we mentioned above, not all code disassembles correctly, and for such cases a debugger is required. The bottom line is that both a disassembler and a debugger are required for any serious SRE task.

The necessary technical skills required for SRE include a working knowledge of the target assembly language and some experience with the necessary tools—primarily a debugger. For Windows, some knowledge of the Windows Portable Executable, or PE, file format is also important [236]. These skills are beyond the scope of this book—see [99] or [161] for more information. Below we'll restrict our attention to simple SRE examples. These examples illustrate the concepts, but do not require any significant knowledge of assembly, any knowledge of the PE file format, etc.

Finally, SRE requires boundless patience and optimism, since the work can be extremely tedious and labor intensive. There are few automated tools, which means that SRE is essentially a manual process that requires many long hours spent slogging through assembly code. From Trudy's perspective, however, the payoff can be well worth the effort.

[1] Here, "patch" means that we directly modify the binary without recompiling the code. Note that this is a different meaning than "patch" in the context of security patches that are applied to code.

12.2.1 Reversing Java Bytecode

Before we consider a "real" SRE example, let's take a quick look at a Java example. When you compile Java source code, it's converted into bytecode and this bytecode is executed by the the Java virtual machine, or JVM. In comparison to, say, the C programming language, the advantage of Java's approach is that the bytecode is more or less machine independent, while the primary disadvantage is a loss of efficiency.

When it comes to reversing, Java bytecode makes Trudy's life much easier. A great deal more information is retained in bytecode than native code, so it is possible to decompile bytecode with great accuracy. There are tools available that will convert Java bytecode into Java source code, and the resulting source code is likely to be very similar to the original source code. There are tools available to obfuscate Java, thereby making Trudy's job more challenging, but none are particularly strong—even highly obfuscated Java bytecode is generally easier to reverse than un-obfuscated machine code.

For example, consider the Java program in Figure 12.1. Note that this program computes and prints the first n Fibonacci numbers, where n is specified by the user.

```
import java.io.*;

public class Fibo
{
    /** Prompt user for a value of n, then
        print n Fibonacci numbers
      */
    public static void main(String[] args) throws IOException {
        BufferedReader rd = new BufferedReader (
                new InputStreamReader(System.in));
        System.out.print("Enter value of n: ");
        String ns = rd.readLine();
        int n = Integer.parseInt(ns);
        int p = 0, c = 1, a;
        while (n-- > 0) {
            System.out.println(c);
            a = p + c;
            p = c;
            c = a;
        }
    }
}
```

Figure 12.1: Java Program

The program in Figure 12.1 was compiled into bytecode and the resulting class file was decompiled using Fernflower, an online tool [110]. This decompiled Java file appears in Figure 12.2.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Fibo
{
    public static void main(String[] var0) throws IOException {
        BufferedReader var1 = new BufferedReader(
                new InputStreamReader(System.in));
        System.out.print("Enter value of n: ");
        String var2 = var1.readLine();
        int var3 = Integer.parseInt(var2);
        int var4 = 0;
        int var6;
        for(int var5 = 1; var3-- > 0; var5 = var6) {
            System.out.println(var5);
            var6 = var4 + var5;
            var4 = var5;
        }
    }
}
```

Figure 12.2: Decompiled Java Program

Note that the original Java source in Figure 12.1 is almost identical to the decompiled Java code in Figure 12.2. The significant differences are that the comments have been lost and the variable names have changed. These differences make the decompiled program slightly more difficult to understand than the original. Nevertheless, Trudy would certainly prefer to decipher code like that in Figure 12.2 rather than deal with assembly code.[2]

As mentioned above, there are tools to obfuscate Java. These tools can obfuscate the control flow and data, insert junk code, and so on. It is even possible to encrypt the bytecode. However, none of these tools seem to be particularly strong—see the homework problems for some examples.

12.2.2 SRE Example

The native code SRE example that we'll consider only requires the use of a disassembler and a hex editor. We'll disassemble the executable to understand the code. Then we'll use the hex editor to patch the code to change its behavior. It's important to realize that this is a very simple example—to do SRE in the real world, a debugger would certainly be required.

For our SRE example, we'll consider code that requires a serial number. The attacker Trudy doesn't know the serial number, and when she guesses (incorrectly) she obtains the results in Figure 12.3.

[2]If you don't believe it, take a look at the next section.

Figure 12.3: Serial Number Program

Trudy could try to brute force guess the serial numbers but that's unlikely to succeed. Being a dedicate reverser, Trudy decides the first thing she'll do is to disassemble `serial.exe`. A small part of the resulting IDA Pro disassembly appears in Figure 12.4.

```
.text:00401003    push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008    call    sub_4010AF
.text:0040100D    lea     eax, [esp+18h+var_14]
.text:00401011    push    eax
.text:00401012    push    offset aS       ; "%s"
.text:00401017    call    sub_401098
.text:0040101C    push    8
.text:0040101E    lea     ecx, [esp+24h+var_14]
.text:00401022    push    offset aS123n456 ; "S123N456"
.text:00401027    push    ecx
.text:00401028    call    sub_401060
.text:0040102D    add     esp, 18h
.text:00401030    test    eax, eax
.text:00401032    jz      short loc_401045
.text:00401034    push    offset aErrorIncorrect ; "Error! Incorrect serial number.
.text:00401039    call    sub_4010AF
```

Figure 12.4: Serial Number Program Disassembly

The line at address `0x401022` in Figure 12.4 indicates that the correct serial number is `S123N456`. Trudy tries this serial number and finds that it is indeed correct, as indicated in Figure 12.5.

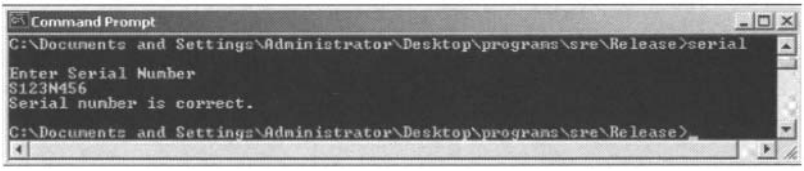

Figure 12.5: Correct Serial Number

But Trudy suffers from short-term memory loss, and she has particular trouble remembering serial numbers. Therefore, Trudy would like to patch the executable `serial.exe` so that she doesn't need to remember the serial number. Trudy looks again at the disassembly in Figure 12.4, and she notices that the `test` instruction at address `0x401030` is significant due to the jump instruction, `jz` at `0x401032` that immediately follows. That is, if the jump occurs, the program will jump elsewhere, bypassing the error message. This has to be good, since Trudy doesn't want to see "Incorrect serial number."

At this point, Trudy must rely on her knowledge of assembly code (or her ability to Google for such knowledge). The instruction `test eax,eax` computes a binary AND of the `eax` register with itself. Depending on the result, this instruction causes various flag bits to be set. One of these flag bits is the zero flag, which is set if `test eax,eax` results in zero. That is, the instruction `test eax,eax` causes the zero flag to be set to one provided that `eax` AND `eax` is zero. With this in mind, Trudy might want to consider ways to force the zero flag bit to be set so that she can bypassing the dreaded "Incorrect serial number" message.

There are many possible ways for Trudy to patch the code. But, whatever approach is used, care must be taken or else the resulting code will not behave as expected. Trudy must take care to only replace bytes. In particular, Trudy cannot insert additional bytes or remove any bytes, since doing so would cause subsequent instructions to be misaligned, that is, the instructions would not align properly, which would almost certainly cause the program to crash.

Trudy decides that she will try to modify the `test` instruction so that the zero flag bit will always be set. If she can accomplish this, then the remainder of the code can be left unchanged. After some thought, Trudy realizes that if she replaces `test eax,eax` with `xor eax,eax`, then the zero flag bit will alway be set to one. This works regardless of what is in the `eax` register, since whenever something is XORed with itself, the result is zero, which will cause the zero flag bit to be set to one. Trudy should then be able to bypass the "Incorrect serial number" message, regardless of which serial number she enters at the prompt.

So, Trudy has determined that changing `test` to `xor` will cause the program to behave as she wants. However, Trudy still needs to determine whether she can actually patch the code to make this change without causing any unwanted side effect. In particular, she must be careful not to insert or delete bytes.

Trudy next examines the bits of the `exe` file (in hex) at address `0x401030` and she observes the results displayed in Figure 12.6, which tells her that `test eax,eax` is, in hex, `0x85C0`.... Relying on her favorite assembly code reference manual, Trudy learns that `xor eax,eax` is, in hex, `0x33C0`.... Trudy realizes she's in luck, since she only needs to change one byte in the executable to make her desired change. Again, it's crucial that she does not need to insert or delete any bytes, as doing so would almost certainly cause the resulting code to fail.

```
.text:00401010  04 50 68 84 80 40 00 E8-7C 00 00 00 6A 08 8D 4C
.text:00401020  24 10 68 78 80 40 00 51-E8 33 00 00 00 83 C4 18
.text:00401030  85 C0 74 11 68 4C 80 40-00 E8 71 00 00 00 83 C4
.text:00401040  04 83 C4 14 C3 68 30 80-40 00 E8 60 00 00 00 83
```

Figure 12.6: Hex View of `serial.exe`

Trudy then uses her favorite hex editor to patch `serial.exe`. Since the addresses in the hex editor won't necessarily match those in the disassembler, she searches through `serial.exe` to find the bits `0x85C07411684C`, as can be seen in Figure 12.6. Since this is the only occurrence of the bit string in the file, she knows this is the right location. She then changes the byte `0x85` to `0x33` and she saves the resulting file as `serialPatch.exe`.

Note that in OllyDbg, for example, patching the code is easier, since Trudy just needs to change the `test` instruction to `xor` in the debugger and save the result. That is, no hex editor is required. In any case, a comparison of the original and the patched executables appears in Figure 12.7.

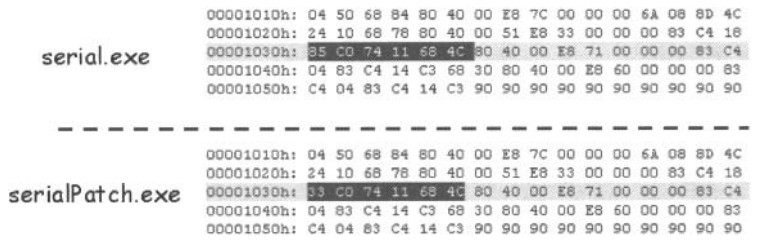

Figure 12.7: Hex View of Original and Patched

Trudy then executes the patched code `serialPatch.exe` and enters an incorrect serial number. The results in Figure 12.8 show that the patched program accepted an incorrect serial number.

Figure 12.8: Patched Executable

Finally, we've disassembled both `serial.exe` and `serialPatch.exe` with the comparison given in Figure 12.9. These snippets of code show that the patching achieved its desired results.

Kaspersky's book [161] is a good source for more information on SRE techniques and the book [233] has a readable introduction to some aspects of SRE. However, the best SRE book available is Eilam's [99]. There are many online SRE resources, perhaps the best of which is at [57].

Next, we'll briefly consider ways to make SRE attacks more difficult. Although it's impossible to prevent such attacks on an open system such as a PC, we can make life more difficult for Trudy. A good, but dated, source of information on anti-SRE techniques is [53].

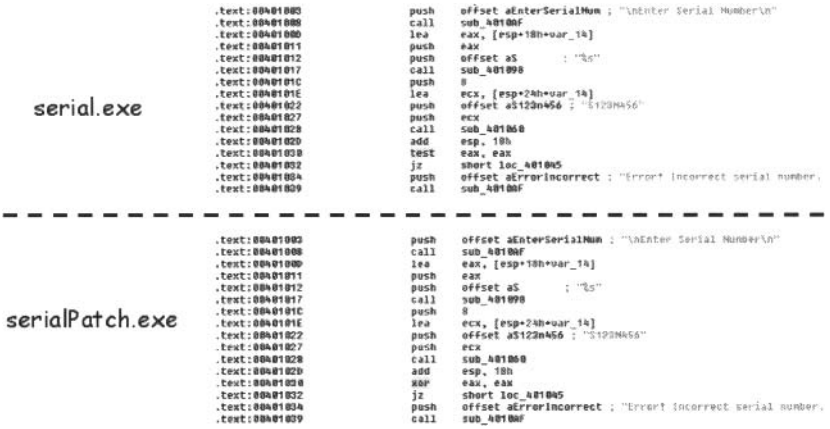

Figure 12.9: Disassembly of Original and Patched

First, we'll consider anti-disassembly techniques, that is, techniques that can be used to confuse a disassembler. Our goal here is to give the attacker an incorrect static view of the code or, better yet, no static view at all. Below, we'll also consider anti-debugging techniques that can be used to obscure the attacker's dynamic view of the code. Then in Section 12.2.5 we'll discuss some tamper-resistance techniques that can be applied to software to make the code more difficult for an attacker to understand and therefore more difficult to patch.

12.2.3 Anti-Disassembly Techniques

There are several well-known anti-disassembly methods.[3] For example, it's possible to encrypt the executable file—when the `exe` file is in encrypted form, it can't be disassembled correctly. But there is a chicken and egg problem here that is similar to the situation that occurs with encrypted viruses. That is, the code must be decrypted before it can be executed. A clever attacker can use the decryption code to gain access to the decrypted executable.

Another simple, but not too effective, anti-disassembly trick is false disassembly [317] which is illustrated in Figure 12.10. In this example, the top part of the figure indicates the actual flow of the program, while the bottom part indicates the false disassembly that will occur if the disassembler is not too smart. In the top part of Figure 12.10, the second instruction causes the program to jump over the junk, which consists of invalid instructions. If a disassembler tries to disassemble these invalid instructions, it will get confused

[3]Your verbose author was tempted to call this section "anti-disassemblymentarianism." Fortunately, he resisted the temptation.

and it may even incorrectly disassemble many instructions beyond the end of the junk, since the actual instructions are not aligned properly. However, if Trudy carefully studies this false disassembly, she will eventually realize that `inst 2` jumps into the middle of `inst 4`, and she can then undo the effects. In fact, quality disassemblers will not be seriously confused by such a simple trick, but slightly more complex examples can have some limited effect.

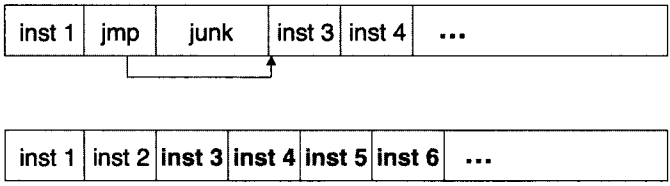

Figure 12.10: False Disassembly

A more sophisticated anti-disassembly trick is self-modifying code. As the name suggests, self-modifying code modifies its own executable in real time [61]. This is a highly effective way to confuse a disassembler, but it's also likely to confuse the developers, since it's difficult to implement, highly error prone, and well-nigh impossible to maintain. Another supposed anti-disassembly approach is discussed in [19].

12.2.4 Anti-Debugging Techniques

There are several methods that can be used to make debugging more difficult. Since debuggers use specific debug registers, a program can monitor the use of these registers and stop (or misbehave) if they are used. That is, a program can monitor for inserted breakpoints, which is a telltale sign of a debugger.

Debuggers don't handle threads well so when properly implemented, interacting threads can offer a relatively strong means for confusing a debugger. In [338] it is shown that by introducing "junk" threads and intentional deadlock among some of these, only a small percentage of the useful code is ever visible in OllyDbg.[4] Furthermore, the code that is visible varies with each run in an unpredictable way. The overhead associated with this approach is fairly high, so it would not be appropriate for the entire code base of a large application. However, this technique could be applied to protect a highly sensitive code section such as that used for entering and checking a serial number.

There are many other debugger-unfriendly tricks, most of which are highly debugger-specific. For example, one anti-debugging technique is illustrated

[4]This does not mean that OllyDbg is a bad debugger—this same trick confuses other popular debuggers at least as much as it confuses OllyDbg.

in Figure 12.11. The top part of the figure gives the series of instructions that are to be executed. Suppose that for efficiency, when the processor fetches `inst 1`, it also prefetches `inst 2`, `inst 3`, and `inst 4`. Also, suppose that when the debugger is running, it does not prefetch instructions. Then we can take advantage of this difference to confuse the debugger, as illustrated in the bottom half of Figure 12.11, where `inst 1` overwrites the memory location of `inst 4`. When the program is not being debugged, this causes no problem since `inst 1` through `inst 4` are all fetched at the same time. But if the debugger does not prefetch `inst 4`, it will be confused when it tries to execute the junk that has overwritten `inst 4` [317].

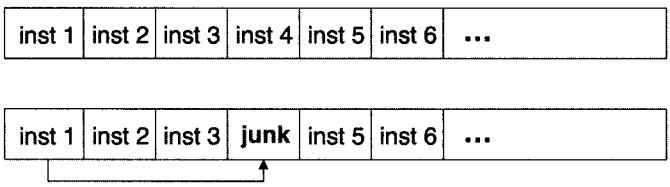

Figure 12.11: Anti-Debugging Example

There are some potential problems with the anti-debugging method in Figure 12.11. First, if the program tries to execute this segment of code more than once (say, within a loop), the junk code will be executed. Also, this code is extremely platform dependent. Finally, if Trudy has enough patience and skill, she will eventually unravel this trick and eliminate its effect.

12.2.5 Software Tamper Resistance

In this section, we discuss several methods that can be employed to make software more tamper resistant. The goal of tamper resistance is to make patching more difficult, either by making the code more difficult to understand or by making the code fail if it's patched. The techniques we'll discuss have been used in practice, but as with most software protection methods, there's little (if any) empirical evidence to support their effectiveness.

12.2.5.1 Guards

It's possible to have a program hash sections of itself as it executes and compare the computed hash values with the known hash values of the original code. If tampering (e.g., patching) occurs, a hash check will fail and the program can take evasive action. These hash checks are sometimes known as guards. Guards can be viewed as a way to make the code fragile in the sense that the code breaks when tampering occurs.

Research has shown that by using guards it's possible to obtain good coverage of software with a minimal performance penalty [54, 145]. But there are some subtle issues. For example, if all guards are identical, then it would be relatively easy for an attacker to automatically detect and remove them. For more information on some issues related to guards, see [268]. Finally, it seems that guards would be ideally suited for use with interacting threads (as discussed above in Section 12.2.4), which could provide a relatively strong defense against tampering.

12.2.5.2 Obfuscation

Another popular form of tamper resistance is code obfuscation. Here, the goal is to make the code difficult to understand. The rationale is that if Trudy can't understand the code, she will have a difficult time patching it. In a sense, code obfuscation is the opposite of good software engineering practices.

As a simple example, spaghetti code can be viewed as a form of obfuscation. There has been much research into more robust methods of obfuscation, and one of the strongest appears to be the *opaque predicate* [64]. For example, consider the following pseudo-code:

```
int x,y;
   ⋮
if((x − y)(x − y) > (x² − 2xy + y²)){...}
```

Notice that the `if` conditional is always false, since

$$(x-y)(x-y) = x^2 - 2xy + y^2$$

for any values of x and y. But an attacker might waste a significant amount of time analyzing the dead code that follows this `if` conditional. While this particular opaque predicate is not particularly opaque, many non-obvious examples have been developed. Again, this technique will not prevent an attack, but it can substantially increase the time and effort required for a successful attack.

Code obfuscation has sometimes been promoted as a powerful general-purpose security technique. In fact, in Diffie and Hellman's original conception of public key cryptography, they suggested a "one-way compiler" (i.e., an obfuscating compiler) as a possible path toward developing such a cryptosystem [90]. However, obfuscation did not turn out to be useful in public key crypto, and recently it has been convincingly argued that obfuscation cannot provide strong protection in the same sense as, say, cryptography [25]. Nevertheless, obfuscation might still have a significant practical benefit in a field such as software protection.

For example, consider a piece of software that is used to determine authentication. Ultimately, authentication is a one-bit decision, regardless of the precise details of the method used. Therefore, somewhere in the authentication software there is, effectively, a single bit that determines whether authentication succeeds or fails. If Trudy can find this bit, she can force authentication to always succeed and thereby break the security. Obfuscation can make Trudy's job of finding this crucial bit into a challenging game of "hide and seek" in software. Obfuscation can, in effect, smear this one bit of information over a large body of code, thereby forcing Trudy to analyze a considerable amount of code. If the time and difficulty required to understand the obfuscated code is sufficiently high, Trudy might give up. If so, the obfuscation has served a useful purpose.

Obfuscation can also be combined with other methods, including any of the anti-disassembly, anti-debugging, or anti-patching techniques discussed above. All of these will tend to increase Trudy's work. However, it is unrealistic to believe that we can drive the cost so high that an army of persistent attackers cannot eventually break our code.

12.2.6 Metamorphism 2.0

The usual practice in software development is to distribute identical copies, or clones, of a particular piece of software. This has obvious benefits with regard to development, maintainability, and so on. But software cloning has some negative security implications. In particular, if an attack is found on any one copy, the exact same attack will work on all copies. That is, the software has no *break once, break everywhere* resistance, or BOBE resistance (this is sometimes rendered as "break once run anywhere," or BORA).

In the previous chapter, we saw that metamorphic software is used by virus writers in to avoid detection. Might a similar technique be used for good instead of evil? For example, suppose we develop a piece of software, but instead of distributing cloned copies, we distribute metamorphic copies. That is, each copy of our software differs internally, but all copies are functionally identical [285]. This is analogous to the metamorphic malware that we discussed in Chapter 11.

Suppose we distribute N cloned copies of a particular piece of software. Then one successful attack breaks all N clones. In other words, this software has no BOBE resistance. On the other hand, if we distribute N metamorphic copies of the software, where each of these N is functionally identical, but they differ in their internal structure, then an attack on one instance will not necessarily work against any other instances. The strength of such an approach depends heavily on how different the non-clones are, but in the best case, N times as much work is required to break all N instances. This is the best possible situation with respect to BOBE resistance.

Thanks to open platforms and SRE, we cannot prevent attacks on software. Arguably, the best we can hope for is increased BOBE resistance. Metamorphism is one possible way to achieve a reasonable level of BOBE resistance.

An analogy is often made between software diversity and genetic diversity in biological systems [61, 115, 114, 194, 221, 230, 231, 277]. For example, if all plants in a field are genetically identical, then one disease can wipe out the entire field. But if the plants are genetically diverse, then one disease will only kill some of the plants. This is essentially the same reasoning that lies behind metamorphic software.

To illustrate the potential benefits of metamorphism, suppose that our software has a common program flaw, say, an exploitable buffer overflow. If we clone this software, then one successful buffer overflow attack will work against all copies of the software. Suppose instead that the software is metamorphic. Then even if the buffer overflow exists in all instances, the same attack will almost certainly not work against many of the instances, since buffer overflow attacks are—as we saw in Chapter 11—fairly delicate.

Metamorphic software is an intriguing concept that has been used in some applications [46, 275]. The use of metamorphism raises concerns regarding software development, software upgrades, and so on. Note that metamorphism does not prevent SRE, but it can provide significant BOBE resistance. Metamorphism is best known for its use in malware, but perhaps it's not just for evil anymore.

12.3 Digital Rights Management

Digital rights management, or DRM, provides a good example of the limitations of doing security in software. Most of the topics discussed in the previous sections of this chapter are relevant to the DRM problem.

In this section, we'll discuss what DRM is, and is not. Then we'll describe an actual DRM system designed to protect PDF documents within a corporate environment. We'll also briefly outline a DRM system designed to protect streaming media, and we'll discuss a proposed peer-to-peer application that employs DRM.

12.3.1 What is DRM?

At its most fundamental level, DRM can be viewed as an attempt to provide "remote control" over digital content. That is, we would like to distribute digital content, but we want to retain some control over its use after it has been delivered [121].

Suppose Trudy wants to sell her new book, *For the Hack of It*, in digital form online. There is a huge potential market on the Internet, Trudy can

keep all of the profits, and nobody will need to pay any shipping charges, so this seems like an ideal solution. However, after a few moments of reflection, Trudy realizes that there is a serious problem. What happens if, say, Alice buys Trudy's digital book and then redistributes it for free online? In the worst case, Trudy might only sell one copy [274, 276].

The fundamental problem is that it's trivial to make a perfect copy of digital content and almost as easy to redistribute it. This is a major change from the pre-digital era, when copying a book was costly and redistributing it was difficult. For an excellent discussion of the challenges faced in the digital age compared with those of the pre-digital era, see the paper [31].

In this section, we'll focus on the digital book example. However, similar comments hold for other digital media, including audio and video.

Persistent protection is a buzzword for the ideal level of DRM protection. That is, we want to protect the digital content so that the protection stays with the content after it's delivered. Examples of the kinds of persistent protection restrictions that we might want to enforce on a digital book include the following:

- No copying
- Read once
- Do not open until Christmas
- No forwarding

What can be done to enforce persistent protection? One option is to rely on the honor system, whereby we do not actually force users to obey the rules but instead simply request that they do so. Since most people are good, honest, decent, law-abiding, and trustworthy, we might expect this to work well. Or maybe not.

Perhaps surprisingly, the honor system has actually been tried. Stephen King, the horror novel writer, published his book *The Plant* online in installments [94, 250]. King said that he would only continue to publish installments if a high enough rate of readers paid.

Of the planned seven installments of *The Plant*, only the first six appeared online. Stephen King's spokesman claimed that the rate of payers had dropped so low that Mr. King would not publish the remaining part online, leaving some angry customers who had paid for 6/7ths of a book [250]. Before dismissing the honor system entirely, it's worth noting that shareware essentially follows the honor system model.

Another option is to give up on enforcing DRM on an open platform such as a PC. In the previous section, we saw that SRE attacks render software on a PC vulnerable. Consequently, if we try to enforce persistent protection through software on an open platform, we are likely doomed to failure.

However, the lure of Internet sales has created an interest in DRM, even if it can't be made perfectly robust. We'll also see that companies have an interest in DRM as a way to comply with certain government regulations.

If we decide that it's worthwhile to attempt DRM on a PC, one option is to build a weak software-based system. Several of these have been deployed, and most are extremely weak. For example, such a DRM system for protecting digital documents might be defeated by a user who is knowledgeable enough to operate a screen capture program.

Another option would be to develop a "strong" software-based DRM system. In the next section we'll describe a system that strives for just such a level of protection. This design is based on a real DRM system developed by your multifaceted author for MediaSnap, Inc., as discussed in [275].

A fairly high level of DRM protection can be achieved on a closed system, such as a game system. These systems are very good at enforcing restrictions similar to the persistent protection requirements mentioned above. There have been efforts to include closed system features in PCs. In large part, this work is motivated by the desire to provide reasonably robust DRM on the PC. We'll return to this topic in Chapter 13 when we discuss Microsoft's Next Generation Secure Computing Base, or NGSCB. In this chapter, we'll only consider software-based DRM.

It is sometimes claimed—or at least strongly implied—that cryptography is the solution to the DRM problem. That this is not the case can easily be seen by considering the generic black box crypto diagram in Figure 12.12, which illustrates a symmetric key system.

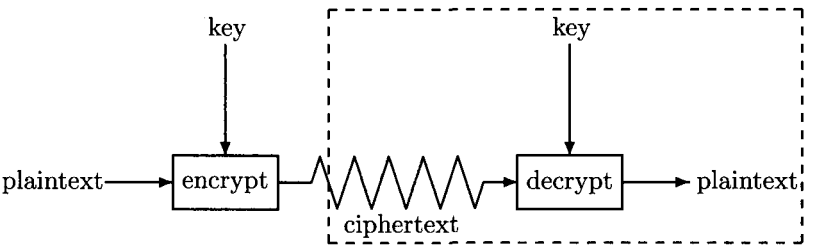

Figure 12.12: Cryptography and DRM

In the standard crypto scenario, the attacker Trudy has access to the ciphertext and perhaps some plaintext and some side-channel information. In the DRM scenario, we are trying to enforce persistent protection on a remote computer. What's more, the legitimate recipient is a potential attacker.

Suppose Trudy is the legitimate recipient of a DRM-protected document. Then Trudy has access to everything within the dashed box in Figure 12.12. In particular, Trudy has access to the key. We certainly can't expect crypto to solve our problem if we give the attacker the key!

With DRM, it's necessary to use encryption so that the data can be securely delivered, and so that Trudy can't trivially remove the persistent protection. But if Trudy is clever, she won't attack the crypto directly. Instead, she will try to find the key, which is hidden somewhere in the software (or at least available to the software at some point in the process). One of the fundamental problems in DRM can be reduced to the problem of playing hide and seek with a key in software [266].

Out of necessity, software-based DRM systems rely largely on *security by obscurity*, that is, the security resides in the fact that Trudy doesn't completely understand the system. In a sense, this is the opposite of Kerckhoffs' Principle. Security by obscurity is generally considered a derogatory term in the security field, since once the obscurity is gone, so is the security. However, in software-based DRM, there is often no other viable option.

Software obfuscation and the other techniques discussed in the previous section are examples of security by obscurity. It's always preferable not to rely on security by obscurity, but, when there is no other option, then we need to consider whether we can derive any useful measure of security from some clever application of obscurity.[5]

Current DRM systems also rely heavily on secret designs, in clear violation of the spirit of Kerckhoffs' Principle. Of course, this is partly due to the reliance on obscurity, but even a general overview of the security architecture is unavailable for most DRM systems, unless it has been provided by some outside source. For example, details on Apple's Fairplay DRM system were not available from Apple, but can be found, for example, in [313].

There is a fundamental limit on the effectiveness of any DRM system, since the so-called *analog hole* is always present. That is, when the content is rendered, it can be captured in analog form—for example, when digital music is played, it can be recorded using a microphone, regardless of the strength of the DRM protection. Similarly, a digital book can be captured in unprotected form using a digital camera to photograph the pages displayed on a computer screen. Such attacks are outside the boundaries of a DRM system.

Another interesting feature of DRM is the degree to which human nature matters. For software-based systems, it's clear that absolute DRM security is impossible, so the challenge is to develop something that might work in practice. Whether this is possible or not depends heavily on the context, as we'll see in the examples discussed below. The bottom line is that DRM is not strictly a technical problem. While this is also true of many security

[5]In spite of its bad name, security by obscurity is used surprisingly often in the real world. For example, system administrators often rename important system files so that they are more difficult for an attacker to locate. If Trudy breaks into the system, it will take her some time to locate these important files, and the longer it takes, the better chance we have of detecting her presence. So, it does make sense to use obscurity in situations such as this.

topics (passwords, the MiM "attack" on SSL, etc.), it's more obvious in DRM than in many other areas.

We've mentioned several times that strong software-based DRM is impossible. Let's be explicit as to why this is the case. From the previous SRE sections, it should be clear that we can't really hide a secret in software, since we can't prevent SRE. A user with full administrator privilege can eventually break any anti-SRE protection and thereby attack DRM software that is trying to enforce persistent protection. In other words, SRE is the "killer app" for attacking software-based DRM.

Next we describe a real-world DRM system designed to protect PDF documents. Then we discuss a system designed to protect streaming media, another system designed for a P2P environment, and, finally, the role of DRM within a corporate environment. Other DRM systems are described in [241] and [314].

12.3.2 A Real-World DRM System

The information in this section is based on a DRM system designed and developed by MediaSnap, Inc., a small Silicon Valley startup company. The system is intended for use with digital documents that will be distributed via email.

There are two major components to the MediaSnap DRM systems, a server component that we'll call the Secure Document Server, or SDS, and the client software, which is a software plugin to the Adobe PDF reader.

Suppose Alice wants to send a DRM-protected document to Bob. Alice first creates the document, then attaches it to an email. She selects the recipient, Bob, in the usual way, and she uses a special pull down menu on her email client to select the desired level of persistent protection. She then sends the email.

The entire email, including any attachments, is converted to PDF and it is then encrypted (using standard crypto techniques) and sent to the SDS. It is the SDS that applies the desired persistent protection to the document. The SDS then packages the document so that only Bob can access it using his client DRM software—it is the client software that will attempt to enforce the persistent protection. The resulting document is then emailed to Bob. This process is illustrated in Figure 12.13.

A key is required to access the DRM-protected document, and this key is stored on the SDS. Whenever Bob wants to access the protected document, he must first authenticate to the SDS and only then will the key be sent from the SDS to Bob. Once Bob gets the key, he can access the document, but only through the DRM software. This process is illustrated in Figure 12.14.

There are security issues both on the server side and on the client side. The SDS must protect keys and authenticate users, and it must apply the

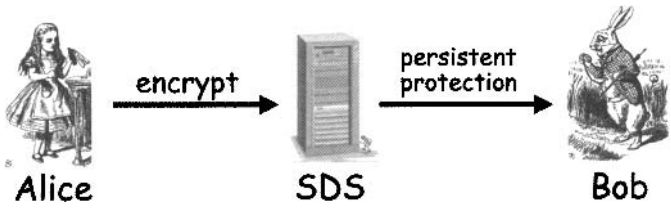

Figure 12.13: DRM for PDF Documents

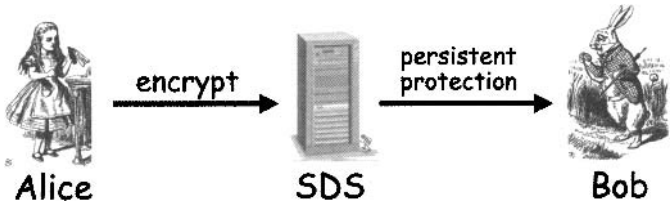

Figure 12.14: Accessing Protected Documents

required persistent protection to documents. The client software must protect keys, authenticate users, and enforce the persistent protection, all while operating in a potentially hostile environment. The SDS resides at corporate headquarters and is relatively secure. The DRM client software, on the other hand, is readily available to any attacker. The discussion below concerns the client software.

The high-level design of the client software is illustrated in Figure 12.15. The software has an outer layer that attempts to create a tamper-resistant barrier. This includes anti-disassembly and anti-debugging techniques, some of which were discussed above. For example, the executable code is encrypted, and false disassembly is used to protect the part of the code that performs the decryption. In addition, the executable code is only decrypted in small slices so that it's more difficult for an attacker to obtain the entire code in plaintext form.

The anti-debugging technique is fairly sophisticated, although the basic idea is simply to monitor for the use of the debug registers. One obvious attack on such a scheme is essentially a man-in-the-middle, where the attacker debugs the code but responds to the anti-debugging software in such a way that it appears no debugger is running.

We know from the previous section that tamper-resistance techniques can delay an attacker, but they can't prevent a persistent attacker from eventual success. The software inside the tamper-resistant layer is heavily obfuscated to further delay an attacker who has penetrated the tamper-resistant outer layer.

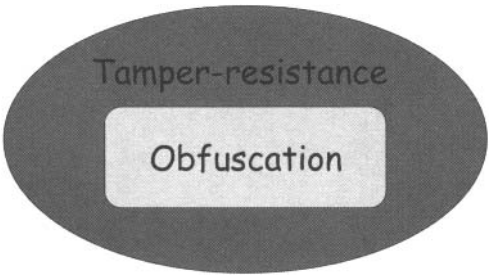

Figure 12.15: DRM Software Design

The obfuscation is applied to security critical operations, including key management, authentication, and cryptography. The authentication information is cached, since we don't want to ask the user to repeatedly enter a password (or other means of authentication). Each time the authentication data is cached, it is cached in a different location in memory and in a different form.

The digital content is encrypted using the Advanced Encryption Standard (AES) block cipher. Unfortunately, standard crypto is difficult to obfuscate since the algorithms are well known and the implementations are standardized for efficiency and to prevent implementation errors. As a result, the MediaSnap system also employs a "scrambling" algorithm, which is essentially a proprietary cipher. This scrambling is used in addition to—and not in place of—a strong cipher, so there is no violation of Kerckhoffs' Principle.

The scrambling algorithm, which is itself obfuscated, presents a much more substantial SRE challenge than a standard cipher, such as AES. The keys are also obfuscated by splitting them into multiple parts and hiding some parts in data and other parts in code. In short, the MediaSnap system employs multiple layers of obfuscation.

Another security feature implemented by the system is an anti-screen capture technique, which is somewhat analogous to the anti-debugging technique mentioned above. Digital watermarking is also employed. As we learned in Chapter 5, watermarking is designed to provide the ability to trace stolen content. However, in practice, watermarking has proven to be of relatively limited value, particularly if the attacker knows the watermarking scheme.

The MediaSnap DRM software employs metamorphism for BOBE resistance. The metamorphism is implemented in several places, most notably in the scrambling algorithms. We'll have more to say about this below when we discuss a DRM application designed to protect streaming media.

The MediaSnap DRM system employs a wide variety of software protection techniques. It is almost certainly one of the most advanced software-based DRM systems ever attempted. The only significant protection mecha-

nism not employed is the guards or "fragilization" technique discussed above, and the only reason guards are not used is that they're not easily incorporated with encrypted executable code.

One major security concern that we did not yet mention is the role of the operating system. In particular, if we can't trust the operating system to behave correctly, then our DRM client software can be undercut by attacks on the operating system. The topic of trusted operating systems is the focus of the next chapter.

12.3.3 DRM for Streaming Media

Suppose we want to stream digital audio or video over the Internet, and this digital media is to be viewed in real time. If we want to charge money for this service, how can we protect the content from capture and redistribution? This sounds like a job for DRM. The DRM system we describe here follows the design given in [282].

Possible attacks on streaming media include spoofing the stream between the endpoints, man-in-the-middle, replay or redistribution of the data, and capturing the plaintext at the client. We are concerned primarily with the latter attack. The threat here arises from unauthorized software that is used to capture the plaintext stream on the client.

The most innovative feature of our proposed design is the use of scrambling algorithms, which are encryption-like algorithms, as described in the previous section. We'll assume that we have a large number of distinct scrambling algorithms at our disposal and we'll use these to achieve a significant degree of metamorphism.

Each instance of the client software comes equipped with a large number of scrambling algorithms included. Each client has a distinct subset of scrambling algorithms chosen from a master set of all scrambling algorithms, and the server knows this master set. The client and server must negotiate a specific scrambling algorithm to be used for a particular piece of digital content. We'll describe this negotiation process below.

We'll also encrypt the content so that we don't need to rely on the scrambling algorithm for cryptographic strength. The purpose of the scrambling is metamorphism—and BOBE resistance—not cryptographic security.

The data is scrambled and then encrypted on the server. On the client, the data must be decrypted and then de-scrambled. The de-scrambling occurs in a proprietary device driver, just prior to rendering the content. The purpose of this approach is to keep the plaintext away from the attacker, Trudy, until the last possible moment prior to rendering.

In the design discussed here, Trudy is faced with a proprietary device driver and each copy of the software has a unique set of hardcoded scrambling algorithms. Therefore, Trudy is faced with a significant SRE challenge and

each copy of the client software presents a distinct challenge. Consequently, the overall system should have good BOBE resistance.

Suppose the server knows N different scrambling algorithms, denoted $s_0, s_1, \ldots, s_{N-1}$. Each client is equipped with a subset of these algorithms. For example, a particular client might have the scrambling algorithms

$$\text{LIST} = \{s_{12}, s_{45}, s_2, s_{37}, s_{23}, s_{31}\}.$$

This LIST is stored on the client as E(LIST, K_{server}), where K_{server} is a key that only the server knows. The primary benefit of this approach is that the database that maps clients to their scrambling algorithms is distributed among the clients, eliminating a potential burden on the server. Notice that this approach is reminiscent of the way Kerberos uses TGTs to manage security-critical information.

To negotiate a scrambling algorithm, the client sends its LIST to the server. The server then decrypts the LIST and chooses one of the algorithms that is built into the client. The server must then securely communicate its scrambling algorithm choice to the client. This process is illustrated in Figure 12.16, where the server has selected the mth scrambling algorithm on the client's LIST. Here, the key K is a session key that has been established between the client and server.

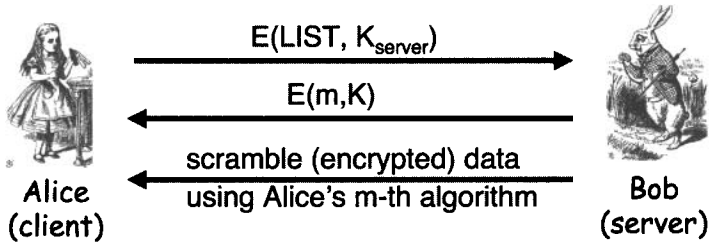

Figure 12.16: Scrambling Algorithm Selection

The metamorphism provided by the scrambling algorithms is deeply embedded in the system and tied to all of the data. Furthermore, if the server knows that a particular scrambling algorithm is broken, the server won't select it. And if a particular client has too many broken algorithms, the server will force a software upgrade before agreeing to distribute the content.

The server can also distribute the client software (or some crucial component of it) immediately prior to distributing the content. This would make it more difficult for Trudy to capture the streamed media in real time, due to the limited time available to attack the software. Of course, Trudy could record the stream and then attack the software at her leisure. However, in many situations, an attack that is not close to real time would be of little concern.

Since the scrambling algorithms are unknown to the attacker, they require a significant effort to reverse engineer, whereas a standard crypto algorithm does not need to be reverse engineered at all—the attacker only needs to find the key. As we mentioned above, it could be argued that such use of scrambling algorithms is just security by obscurity. But in this particular application, it appears to be of some value since it improves BOBE resistance.

12.3.4 DRM for a P2P Application

Today, much digital content is delivered via peer-to-peer, or P2P, networks. For example, such networks contain large amounts of illegal, or pirated, music. The following scheme is designed to gently coerce users into paying a small fee for legal content that is distributed over a P2P network. Note that this P2P network may contain large amounts of illegal content in addition to the legal content.

The scheme we describe here is based on the work of Exploit Systems [108]. But before we discuss this application in detail, let's briefly review how a P2P network works.

Suppose Alice has joined a P2P network, and she requests some music, say, "Relay" by The Who. Then a query for this song floods through the network, and any peer who has the song—and is willing to share it—responds to Alice. This is illustrated in Figure 12.17. In this example, Alice can choose to download the song from either Carol or Pat.

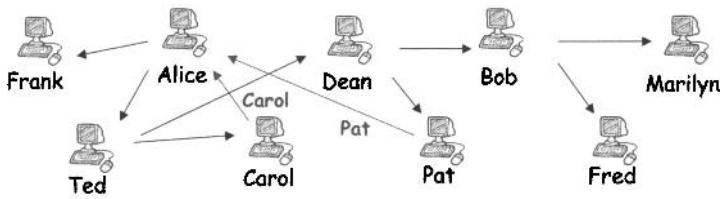

Figure 12.17: P2P Network

Figure 12.18 illustrates the same scenario in a P2P network that includes a special peer that we'll call a *peer offering service*, or POS. The POS acts much like any other peer, except that it has only legal—and DRM-protected—music.

When Alice makes her request on a P2P network with a POS, it appears to her that she has received responses from Bill, Ben, Carol, Joe, and Pat. If Alice selects to download the music from Bill, Ben, or Joe, she will receive DRM protected content for which she will be required to pay a small fee before she can listen to the music. On the other hand, if Alice selects either Carol or Pat, she receives the music for free, just as in the P2P network without the POS.

For the POS concept to work, it must not be apparent to Alice whether a peer is an ordinary peer or a POS peer. In addition, the POS must have a significant percentage of its peers appear in the top ten responses. Let's assume that these technical challenges can be resolved in favor of the POS.

Now suppose Alice first selects Bill, Ben, or Joe. Then after downloading the music and discovering that she must pay, Alice is free to select another peer and, perhaps, another, until she finds one that has pirated (i.e., free) music. But is it worth Alice's time to download the song repeatedly just to avoid paying? If the music is priced low enough, perhaps not. In addition, the legal (DRM-protected) version can offer extras that might further entice Alice to pay a small fee.

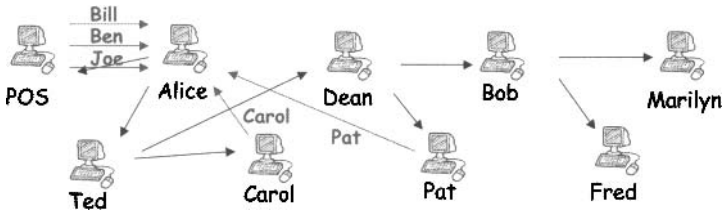

Figure 12.18: P2P Network with POS

The POS idea is clever, since it piggybacks on existing P2P networks. And in the POS scenario, relatively weak DRM is sufficient. As long as it's more trouble for Alice to break the DRM than to click and wait for another download, the DRM has served its purpose.

12.3.5 Enterprise DRM

There are government regulations that require companies to protect certain types of private information and there are similar regulations regarding many types of business records. For example, the Health Insurance Portability and Accountability Act, or HIPAA, requires that companies protect personal medical records. HIPAA stipulates fines of up to $10,000 per incident (i.e., per record) for failing to provide sufficient protection. Companies that deal with medical records often need to make such records accessible to certain employees, but, due to HIPAA, they also must be careful that these records do not leak to unauthorized recipients. DRM can help to solve this problem.

The Sarbanes-Oxley Act, or SOA, requires that companies must preserve certain documents, such as information that might be relevant to insider trading stock violations. Again, DRM could be used here to be sure that such information is protected as required by law. The bottom line is that DRM-like protections are needed by corporations for regulatory compliance.

We refer to this as *enterprise DRM* to distinguish it from the e-commerce scenarios discussed above.

From a technical point of view, the enterprise DRM security requirements are similar to those for e-commerce. But the motivation for enterprise DRM is entirely different, since the purpose is to prevent a company from losing money (due to fines) instead of being an avenue for making money (as in e-commerce). More significantly, the human dimension is completely different. In an enterprise setting the threat of reprisals (getting fired or sued) are far more plausible than in the e-commerce setting. Also, the required level of protection is different. In enterprise DRM, a corporation has likely shown due diligence and thereby complied with the regulations, provided that an active attack on the DRM system is required to break its security. A moderate level of DRM is sufficient in this case. From a technical perspective, enterprise DRM is very much a solvable problem.

In e-commerce, the strength of the DRM system is the predominate concern. But in the enterprise setting, other more mundane issues are more important [286]. For example, policy management is an important concern. That is, it must be easy for an administrator to set policies for individual users, groups, etc. Authentication issues are also significant, since the DRM system must interface with an existing corporate authentication system, and the system must prevent authentication spoofing. From a technical perspective, these are not major obstacles.

DRM for e-commerce and enterprise DRM face similar technical hurdles. But because the human dimension is so different, one is virtually unsolvable (at least for software-based systems), while the other is fairly easy.

12.3.6 DRM Failures

There are far too many examples of failed e-commerce DRM systems to list them all here, but we'll mention a few. One infamous system could be defeated by a felt-tip pen [97], while another was defeated by holding down the shift key while downloading the content [6].

The Secure Digital Music Initiative, or SDMI, is an interesting case. Prior to implementing SDMI on real-world systems, the SDMI Consortium posted a series of challenge problems online, presumably to show how secure their system would be in practice. A group of researchers was able to completely break the security of the SDMI, and for their hard work they were rewarded with the threat of multiple lawsuits. Eventually the attackers' results were published, and they make fascinating reading—particularly with respect to the inherent limitations of watermarking schemes [71].

Major corporations have put forth DRM systems that were easily broken. For example, Adobe eBooks security was defeated [23, 133], and as in the case of SDMI, the attacker's reward consisted of unenforceable legal threats [310].

Another poor DRM system was Microsoft's MS-DRM (version 2). Microsoft violated Kerckhoffs' Principle, which resulted in a fatally flawed block cipher algorithm. The attacker in this case was "Beale Screamer" [29], who avoided legal reprisals, presumably due to his anonymity.

12.3.7 DRM Conclusions

DRM illustrates the limitations of doing security in software, particularly when that software must function in a hostile environment. Such software is vulnerable to attack, and the protection options are extremely limited. In other words, the attacker has nearly all of the advantages.

Tamper-resistant hardware and a trusted operating system can make a significant difference. We'll discuss these topics more in Chapter 13.

In the next section, we shift gears to discuss security issues related to software development. Much of our discussion will be focused through the lens of the open source versus closed source software debate.

12.4 Software Development

The standard approach to software development is to develop and release a product as quickly as possible. While some testing is done, it is almost never sufficient, so the code is patched as flaws are discovered by users.[6] In security, this is known as *penetrate and patch.*

Penetrate and patch is a bad way to develop software in general, and a terrible way to develop secure software. Since it's a security liability, why is this the standard software development paradigm? There is more to it than simply an ethical failing by software developers. In software, whoever is first to market is likely to become the market leader, even if their product ultimately is inferior to the competition. And in the computing world, the market leader tends to dominate more so than in most fields. This first to market advantage creates an overwhelming incentive to sell software before it's been thoroughly tested.

There also seems to be an implicit assumption that if you patch bad software long enough it will eventually become good software. This is sometimes referred to as the *penetrate and patch fallacy* [317]. Why is this a fallacy? For one thing, there is huge body of empirical evidence to the contrary—regardless of the number of service packs applied, software continues to exhibit serious flaws. In fact, patches often add new flaws. And software is a moving target due to new versions, new features, changing environment, new uses, new attacks, and so on.

[6]Note that "patch" has a slightly different meaning here than in the SRE context. Here, it means "to fix bugs," whereas in SRE it refers to a change made directly to the executable code to add, remove, or modify certain features of the software.

Another contributing factor toward the current sorry state of software security is that users generally find it easier and safer to follow the leader. For example, a system administrator probably won't get fired if his system has a serious flaw, provided everybody else has the same flaw. On the other hand, that same administrator might not receive much credit if his system works normally while other systems are having problems.

Yet another major impetus for doing things like everybody else is that administrators and users have more people they can ask for support. Together, these perverse economic incentives are sometimes collectively referred to as *network economics* [14].

Secure software development is difficult and costly. Development must be done carefully, with security in mind from the beginning. And, as we'll make somewhat precise below, an extraordinarily large amount of testing is required to achieve reasonably low bug rates. It's certainly cheaper and easier to let customers do the testing, particularly when there is no serious economic disincentive to do so, and, due to network economics, there is an enormous incentive to rush to market.

Why is there no economic disincentive for flawed software? Even if a software flaw causes major losses to a corporation, the software vendor has no legal liability. Few, if any, other products enjoy a comparable legal status. In fact it's sometimes suggested that holding software vendors legally liable for the adverse effects of their products would be a market-friendly way to improve the quality of software. But software vendors have so far successfully argued that such liability would stifle innovation. In any case, it's far from certain that such an approach would have any serious impact on the overall quality of software. Even if software quality did improve, the cost might be greater than anticipated and there would certainly be some unintended negative consequences.

12.4.1 Open Versus Closed Source Software

We'll look at some of the security problems inherent in software through the prism of the open source versus closed source debate. Some of the conclusions will probably surprise you.

With open source software, the source code is available to users. For example, the Linux operating system is open source. With closed source software, on the other hand, the source code is not available to the general public. Windows is an example of closed source software. In this section, we want to examine the relative security strengths and weaknesses of open source and closed source software.

The primary claimed security advantages of open source software can be summarized as "more eyeballs," that is, more people can look at the code, so fewer flaws should remain undiscovered. This is really just a variant on

Kerckhoffs' Principle, and what self-respecting security person could possibly argue with that?

However, upon closer examination, the benefit of more eyeballs becomes more questionable, at least with respect to software security. First, how many of these eyeballs are looking for security flaws? And how many are focused on the low-level (tedious, boring) parts of the code, which are more likely to harbor security flaws? Also, how many of these eyeballs belong to people who are knowledgable about security—those who would have a realistic chance of discovering subtle security flaws?

Another issue with open source is that attackers can also look for flaws in the source code. Conceivably, an ingenious evil coder might even be able to insert a security flaw into an open source project. While this may sound far-fetched, the Underhanded C Contest shows that it's possible to write evil code that looks innocent [70].

An interesting open source case study is `wu-ftp`. This open source software is of modest size, at about 8,000 lines of code, and it implements a security-critical application (file transfer). Yet this software was widely deployed and in use for ten years before serious security flaws were discovered [317]. More generally, the open source movement appears to have done little to reduce security flaws. Perhaps the fundamental problem is that open source software also follows the penetrate and patch model of development. However, there is some evidence that open source software is significantly less buggy than closed source [84].

If open source software has its security issues, certainly closed source software is worse. Or is it? The security flaws in closed source are not as visible to attackers, which could be viewed as providing some protection (although it could be argued that this is just a form of security by obscurity). But does this provide any significant protection? Given the record of attacks on closed source software, it is clear that many exploits do not require source code—our simple SRE example in Section 12.2 illustrates why this is the case. Although it is possible to analyze closed source code, it's a lot more work than for open source software.

Advocates of open source often cite the *Microsoft fallacy* as a reason why open source software is inherently superior to closed source [317]. This fallacy can be summarized as follows.

1. Microsoft makes bad software.
2. Microsoft software is closed source.
3. Therefore all closed source software is bad.

While it is always tempting to blame everything on Microsoft, this one doesn't hold water. For one thing, it's not logically correct. Perhaps the real issue is the fact that Microsoft follows the penetrate and patch model.

Next, we'll take a little closer look at the security of open source and closed source software. But before we get to that, it's reasonable to ponder why Microsoft software is successfully attacked so often. Is there some fundamental problem with Microsoft software?

Microsoft is obviously a big target for any attacker—an attacker who wants the most bang for the buck is naturally attracted to Microsoft. While there are few exploits against, say, Mac OS X, this almost certainly has more to do with the fact that it receives less attention from hackers (and, not coincidentally, the hacker tools are much less well developed) than any inherent security advantage of OS X. An attack on OS X would do far less damage overall and therefore bring less "glory" to the attacker. Even from the perspective of stealthy attacks, such as botnets, there is much more incentive to attack a big target like Microsoft—numbers matter.

Now let's consider the security implications of open and closed source software from a slightly more theoretical angle. It can be shown that the probability of a security failure after t units of testing is about K/t, where K is a constant, and this approximation holds over a large range of values for t [12]. The constant K is a measure of the initial quality of the software—the smaller K, the better the software was initially. This formula implies that the *mean time between failure*, or MTBF, is given by

$$\text{MTBF} = t/K. \tag{12.1}$$

That is, the average amount of time until some software security flaw rears its ugly head and causes problems is t/K, where t is the amount of time that has been spent testing the software. The bottom line is that software security improves with testing, but it only improves linearly.

The implication of equation (12.1) is bad news for the good guys. For example, to achieve a level of, say, 1,000,000 hours between security failures, software must be tested for (on the order of) 1,000,000 hours.

Is it really true that software only improves linearly with testing? Empirical results have shown that this is the case, and it is the conventional wisdom of many in the software field that this is reality for large and complex software systems [14].

What does equation (12.1) imply about the security of open source versus closed source software? Consider a large and complex open source project. Then we would expect this project to satisfy equation (12.1). Now suppose this same project was instead closed source. Then it would be reasonable to expect that the flaws are harder to find than in the open source case. For simplicity, suppose the flaws are twice as hard to find in the closed source case. Then it might seem that

$$\text{MTBF} = 2t/K. \tag{12.2}$$

If this is correct, closed source software is twice as secure as open source. However, equation (12.2) is not correct, since the closed source testing is only half as effective as in the open source case, that is, we need to test twice as long to expose the same number of bugs. In other words, the closed source software has more security flaws, but they are harder to find. In fact, if the flaws are twice as hard to find, then our testing is only half as effective and we arrive back at equation (12.1). This Zen-like argument shows that, in some sense, the security of open and closed source software is indistinguishable—see [12] for more details.

It might be argued that closed source software has open source alpha testing, where flaws are found at the higher open source rate, since developers have access to the software. This alpha testing is followed by closed source beta testing and use, where customers actually use the software and, effectively, test it in the process. This combination would seem to yield the best of both worlds—fewer bugs due to the open source alpha testing with the remaining bugs harder to find due to the code being closed source. However, in the larger scheme of things, alpha testing is a small part of the total testing, particularly with the pressures to rush to market. Although this argument could, in principle, give an edge to closed source, in practice it's probably not a significant advantage. The surprising conclusion here is that open and closed source software are probably about the same from a security perspective.

12.4.2 Finding Flaws

A fundamental security problem with software testing is that the good guys must find almost all security flaws, whereas Trudy only needs to find one that the good guys haven't yet found. This implies that software reliability is far more challenging in security than in software engineering in general.

An example from [14] nicely illustrates this asymmetric warfare between attacker and defender. Recall that the mean time between faliure is given by MTBF $= t/K$. For the sake of argument, suppose there are 10^6 security flaws in a large and complex software project and assume that for each individual flaw, MTBF $= 10^9$ hours. That is, any specific flaw is expected to show up after about a billion hours of use. Then, since there are 10^6 flaws, we would expect to observe one flaw for every $10^9/10^6 = 10^3$ hours of testing or use.

Suppose that the good guys hire 10,000 testers who spend a total of 10^7 hours testing, and they find, as expected, 10^4 flaws. Evil Trudy, by herself, spends 10^3 hours testing and finds one flaw. Since the good guys found only 1% of the flaws, the chance that they found Trudy's specific bug is only 1%. This is not good. As we've seen in other areas of security, the math overwhelmingly favors the bad guys.

12.4.3 Other Software Development Issues

Software development generally includes the following steps [235]: specify, design, implement, test, review, document, manage, and maintain. Most of these topics are beyond the scope of this book, but in this section, we'll mention a few software development issues that have a significant impact on security.

Secure software development is not easy, as our previous discussion of testing indicates. And testing is only part of the development process. To improve security, much more time and effort are required throughout the entire development process. Unfortunately, there is little or no economic incentive for this today.

Next, we'll briefly discuss the following security-critical software development topics:

- Design
- Hazard analysis
- Peer review
- Testing
- Configuration management
- Postmortem for mistakes

We've already discussed testing, but we'll have more to say about some other testing-related issues below.

The design phase is critical for security since a careful initial design can avoid high-level errors that are difficult—if not impossible—to correct later. Perhaps the most important point is to design security features in from the start, since retrofitting security is difficult, if not impossible. Internet protocols offer an excellent illustration of this difficulty. IPv4, for example, has no built-in security, while the new-and-improved version, IPv6, makes IPSec mandatory. However, the transition to IPv6 is proving slow to nonexistent and, consequently, the Internet remains much less secure than it could be.

Usually an informal approach is used at the design phase, but so-called *formal methods* can sometimes be applied [40]. Using formal methods, it's possible to rigorously prove that a design is correct. Unfortunately, formal methods are generally too difficult to be practical in most real-world situations.

To build secure software, the threats must be considered in advance. This is where the field of *hazard analysis* comes into play. There are several informal ways to approach this problem, such as developing a hazard list containing potential security problems, or simply making a list of "what ifs."

A slightly more systematic approach is Schneier's *attack tree* concept, where possible attacks are organized into a tree-like structure [259]. A nice feature of this approach is that you can prune entire branches of attacks if you can prevent the attacks closer to the root of the tree.

There are several other approaches to hazard analysis, including hazard and operability studies (HAZOP), failure modes and effective analysis (FMEA), and fault tree analysis (FTA) [235]. We'll not discuss these topics here.

Peer review is also a useful tool for improving security. There are three levels of peer review which, from most informal to most formal, are sometimes called *review*, *walk-through*, and *inspection*. Each level of review is useful, and there is good empirical evidence that peer review is effective [235].

Next, we'll discuss testing, but from a different perspective than above in Section 12.4. Testing occurs at different levels of the development process, which can be categorize as follows:

- *Module testing* — Small sections of the code are tested individually.
- *Component testing* — A few modules are combined and tested together.
- *Unit testing* — Many components are combined for testing.
- *Integration testing* — Everything is put everything together and tested as a whole.

At each of these levels, security flaws can be uncovered. For example, features that interact in a new or unexpected way may evade detection at the component level but be exposed during integration testing.

Another way to view testing is based on its purpose. We can define categories as follows:

- *Function testing* — Here, we verify that the system functions as required.
- *Performance testing* — Requirements such as speed and resource use are verified.
- *Acceptance testing* — The customer is actively involved.
- *Installation testing* — Not surprisingly, this is testing done at install time.
- *Regression testing* — Testing that is done after any significant change to the system.

Again, security vulnerabilities can be exposed during any of these types of testing.

Another useful testing technique is *active fault detection*, where instead of simply waiting for a system to fail, the tester actively tries to make it fail. This is the approach that an attacker will follow and it might uncover security flaws that a more passive approach would miss.

An interesting concept is *fault injection*, where faults are inserted into the process, even if there is no obvious way for such a fault to occur. This might, for example, reveal buffer overflow problems that would otherwise go unnoticed if the testing is restricted to expected inputs.

Bug injection can enable testers to obtain an estimate on the number of bugs remaining in code. Suppose we insert 100 bugs into our code and our testers find 30 of these. Further, suppose that in addition to these 30 bugs, our testers find 300 other bugs. Since the testers found 30% of the inserted bugs, it might be reasonable to assume that they also found 30% of the actual bugs. If so, then roughly 700 bugs would remain, after removing all of the discovered bugs and the 70 remaining inserted bugs. Of course, this assumes that the injected bugs are similar to the naturally occurring bugs, which is probably not entirely valid. Nevertheless, bug injection may provide a useful estimate of the number of bugs and, indirectly, the number of security flaws.

A testing case history is given in [235]. In this example, the system had 184,000 lines of code. Flaws were found at the following rates:

- 17.3% were found when inspecting the system design.
- 19.1% were found inspecting component design.
- 15.1% were found during code inspection.
- 29.4% were found during integration testing.
- 16.6% were found during system and regression testing.

The conclusion is that many kinds of testing must be conducted and that overlapping testing is helpful.

Configuration management, that is, how we deal with changes to a system, can also be a security-critical issue. Several types of changes can occur, and these changes can be categorized as follows: *minor changes* are needed to maintain daily functioning, *adaptive changes* are more substantial modifications, while *perfective changes* are improvements to the software, and, finally, *preventive changes*, which are intended to prevent any loss of performance [235]. Any such changes to a system can introduce new security flaws or expose existing flaws, either directly as a result of the new software, or due to interactions with the existing software base.

After identifying and fixing any security flaw, it is important to carefully analyze the flaw. This sort of *postmortem analysis* is the best way to learn from the problem and thereby increase the odds that a similar problem will be avoided in the future. In security, we always learn more when things go wrong than when they go right. If we fail to analyze those cases where we know that things went wrong, then we've missed a significant opportunity. Postmortem analysis may be the most underutilized method in all of security engineering.

As we observed earlier in this chapter, security testing is far more demanding than non-security testing. In the latter case, we need to verify that the system does what it's supposed to, while in security testing we must verify that the system does what it is supposed to and nothing more. That is, there can be no unintended "features," since any such feature provides a potential avenue of attack.

In any realistic scenario, it's almost certainly impossible to do exhaustive testing. Furthermore, the MTBF formula discussed in Section 12.4.1 indicates that an extraordinarily large amount of testing would be required to achieve a high level of security. So, is secure software really as hopeless as it seems? Fortunately, there may be a loophole. If we can eliminate an entire class of potential security flaws with one (or a few) tests, then the statistical model that the MTBF is based on will break down [14]. For example, if we have a test (or a few tests) that enable us to find all buffer overflows, then we can eliminate this entire class of serious flaws with a relatively small amount of work. This is the holy grail of software testing in general, and security testing in particular.

The bottom line on secure software development is that network economics and penetrate and patch are the biggest enemies of secure software. Unfortunately, there is generally little incentive for secure software development, and until that changes, we probably can't expect major improvements in security. In those cases where security is a high priority, it is possible to develop reasonably secure software, but there is most definitely a cost. That is, proper development practices can minimize security flaws, but secure development is a costly and time-consuming proposition.[7] For all of these reasons (and more), you should not expect to see a dramatic improvements in software security anytime soon.

Even with the best software development practices, security flaws will still exist. Since absolute security is almost never possible in the real world, it should not be surprising that absolute security in software is not realistic. In any case, the goal of secure software development—as in most areas of security—is to minimize and manage the risks.

[7]As you probably realize, it's that annoying "no free lunch" thing yet again.

12.5 Summary

In this chapter we showed that security in software is difficult to achieve. We focused on three topics, namely, reverse engineering, digital rights management, and software development.

Software reverse engineering (SRE) illustrates what an attacker can do to software. Even without access to the source code, an attacker can understand and modify your code. Making very limited use of the available tools, we were able to easily defeat the security of a program. While there are things that can be done to make reverse engineering more difficult, as a practical matter, most software is wide open to SRE-based attacks.

We then discussed digital rights management (DRM), which illustrates the futility of attempting to enforce strong security measures through software. After our look at SRE, this should not have come as a surprise.

Finally, we discussed the difficulties involved in secure software development. Although we looked at the problem from the perspective of open source versus closed source software, from any perspective secure software development is extremely challenging. Some elementary math confirms that the attacker has most of the advantages. Nevertheless, it is possible—although difficult and costly—to develop reasonably secure software. Unfortunately, today secure software is the exception rather than the rule.

12.6 Problems

1. Obtain the file `SRE.zip` from the textbook website and extract the Windows executable.

   a. Patch the code so that any serial number results in the message "Serial number is correct!!!" Turn in a screen capture showing your results.

   b. Determine the correct serial number.

2. For the SRE example in Section 12.2.2, we patched the code by changing a `test` instruction to `xor`.

   a. Give at least two ways—other than changing `test` to `xor`—that Trudy could patch the code so that any serial number will work.

   b. Changing the `jz` instruction that appears at address `0x401032` in Figure 12.4 to `jnz` is not a correct solution to part a. Why not?

3. Obtain the file `unknown.zip` from the textbook website and extract the Java class file `unknown.class`.

   a. Use CafeBabe [44] to reverse this class file.

b. Analyze the code to determine what the program does.

4. Obtain the file `Decorator.zip` from the textbook website and extract the file `Decorator.jar`. This program is designed to evaluate a student's application for admission based on various test scores. Applicants applying to medical school must include their score on the MCAT test score, while applicants to law school must include their score on the LSAT test. Applicants to the graduate school (which includes Law and Medicine) must include their score on the GRE test, and foreign applicants must include their score on the TOEFL exam. An applicant is accepted if his or her GPA is above 3.5 and they exceed a set threshold for their required tests (MCAT, LSAT, GRE, TOEFL). Since the school is locate in California, the requirements are more lenient for California residents. This program creates six applicants of which two are not accepted because of their low score. Finally, the program was obfuscated using ProGuard (using only options under the "obfuscation" button, i.e., no shrinking, optimization, etc., were applied); see [58] for a detailed solution to a similar example.

   a. Patch the program so that the two applicants who were not accepted are accepted. Accomplish this by lowering the thresholds in their respective failing categories to the values of their scores.

   b. Using the result from part a, further patch the code so that a California resident who was accepted (in the original program) is now rejected.

5. Obtain the file `encrypted.zip` from the textbook website and extract the file `encrypted.jar`. This application was encrypted using SandMark [63], with the "obfuscate" tab and "Class Encryptor" option selected and, possibly, other obfuscation options.

   a. Generate a decompiled version of this program directly from the obfuscated (and encrypted) code. Hint: Do not attempt to use a cryptanalytic attack to break the encryption. Instead, look for an unencrypted class file. This is a custom class loader that decrypts the encrypted files before they are executed. Reverse this custom class loader and modify it so that it prints out the class files in plaintext.

   b. How could you make this encryption scheme more difficult to break?

6. Obtain the file `deadbeef.zip` from the textbook website and extract the C source file `deadbeef.c`.

a. Modify the program so that it tests for a debugger using the Windows function `IsDebuggerPresent`. The program should silently terminate if a debugger is detected, whether or not the correct serial number is entered.

b. Show that you can determine the serial number using a debugger, in spite of the `IsDebuggerPresent()` function. Briefly explain how you were able to bypass the `IsDebuggerPresent()` check.

7. Obtain the file `mystery.zip` from the textbook website and extract the Windows executable `mystery.exe`.

   a. What is the output when you run the program with each of the following usernames, assuming an incorrect serial number in each case?

      i. mark
      ii. markstamp
      iii. markkram

   b. Analyze the code to determine all restrictions, if any, on valid usernames. You will need to disassemble and/or debug the code.

   c. This program uses an anti-debugging technique, namely, the Windows system function `IsDebuggerPresent()`. Analyze the code to determine what the program does in case a debugger is detected. Why is this better than simply terminating the program?

   d. Patch the program so that you can debug it. That is, you need to nullify the effect of `IsDebuggerPresent()`.

   e. By debugging the code, determine the corresponding valid serial number for each valid username that appears in part a. Hint: Debug the program and enter a username along with any serial number. At some point the program will compute the valid serial number corresponding to the entered username—it does this so that it can compare to the entered serial number. If you set a breakpoint at the correct location, the valid serial number will be stored in a register, which you can then observe.

   f. Create a patched version of the code, `mysteryPatch.exe` that accepts any username/serial number pair.

8. Obtain `mystery.zip` from the textbook website and extract the Windows executable `mystery.exe`. As mentioned in Problem 7, part e, the program contains code that generates a valid serial number corresponding to any valid username. Such an algorithm is known as a key generator, or simply a *keygen*. If Trudy has a functioning copy of the keygen algorithm, she can generate an unlimited number of valid

username/serial number pairs. In principle, it would be possible for Trudy to analyze a keygen algorithm and write her own (functionally equivalent) standalone keygen program from scratch. However, keygen algorithms are generally complex, making such an attack difficult in practice. But all is not lost (at least from Trudy's perspective). It is often possible—and relatively simple—to "rip" the keygen algorithm from a program. That is, an attacker can extract the assembly code representing the keygen algorithm and embed it directly in a C program, thereby creating a standalone keygen utility, without having to understand the details of the algorithm.

a. Rip the keygen algorithm from `mystery.exe`, that is, extract the keygen assembly code and use it directly in your own standalone keygen program. Your program must take any valid username as input and produce the corresponding valid serial number. Hint: In Visual C++ assembly code can be embedded directly in a C program by using the `asm` directive. You may need to initialize certain register values to make the ripped code function correctly.

b. Use your program from part a to generate a serial number for the username `markkram`. Verify that your serial number is correct by testing it in the original `mystery.exe` program.

9. This problem deals with software reverse engineering (SRE).

   a. Suppose debugging is impossible. Is SRE still possible?

   b. Suppose disassembly is impossible. Is SRE still possible?

10. How can the the anti-debugging technique illustrated in Figure 12.11 be implemented so that it also provides anti-disassembly protection?

11. Why are guards incompatible with encrypted object code?

12. Recall that an opaque predicate is a "conditional" that is actually not a conditional. That is, the conditional always evaluates to the same result, but it is not obvious that this is the case.

    a. Why is an opaque predicate a useful defense against reverse engineering attacks?

    b. Give an example—different from that given in the text—of an opaque predicate based on a mathematical identity.

    c. Give an example of an opaque predicate based on an input string.

13. The goal of this problem is to show that you can convert any conditional into an opaque predicate.

a. Given the conditional

```
if(a < b)
   // do something
else
   // do something else
```

slightly modify the `if` statement so that the `do something` branch always executes.

b. Explain why your solution to part a will work in general.

c. How stealthy is your approach, that is, how difficult would it be for an attacker to (automatically) detect your opaque predicates? Could you make your approach stealthier?

14. Opaque predicates have been proposed as a method for watermarking software [18, 212].

a. How might such a watermarking technique be implemented?

b. Consider possible attacks on such a watermarking scheme.

15. Describe in detail one anti-disassembly method not discussed in this chapter.

16. Describe in detail one anti-debugging method not discussed in the text.

17. Consider a DRM system implemented in software on a PC.

a. Define persistent protection.

b. Why is encryption necessary, but not sufficient, to provide persistent protection?

18. Consider a DRM system implemented in software on a PC. As discussed in the text, such systems are inherently insecure. Suppose that in an alternate universe such a system could be made highly secure.

a. How would such a system benefit copyright holders?

b. How could such a system be used to enhance privacy? Give a concrete example.

19. Suppose that it's impossible to patch some particular software that implements DRM protection. Is the DRM system then secure?

20. Some DRM systems have been implemented on open systems and some have been implemented in closed systems.

a. What is the primary advantage of implementing DRM on a closed system?

b. What is the primary advantage to implementing DRM on an open platform?

21. Once a user authenticates, it is sometimes desirable to have the program keep this authentication information available, so that we do not need to bother the user to authenticate repeatedly.[8]

    a. Devise a method for a program to cache authentication information, where the information is stored in a different form each time it's cached.

    b. Is there any security advantage to your approach in part a, as compared to simply storing the information the same each time?

22. Above, we discuss break-once, break-everywhere (BOBE) resistance.

    a. Why is BOBE resistance desirable for software in general, and DRM systems in particular?

    b. In the text, it is argued that metamorphism can increase BOBE resistance. Discuss one other method that could be used to increase BOBE resistance.

23. In [266], it's shown that keys are easy to find when hidden in data, since keys are random and most data is not.

    a. Devise a more secure method for hiding a key in data.

    b. Devise a method for storing a key K in data and in software. That is, both the code and the data are required to reconstruct the key K.

24. In an analogy to genetic diversity in biological systems, it is sometimes argued that metamorphism can increase the resistance of software to certain types of attacks, such as buffer overflow.

    a. Why should metamorphic software be more resistant to buffer overflow attacks? Hint: See [281].

    b. Discuss other types of attacks that metamorphism might help to prevent.

    c. From a development perspective, what difficulties does metamorphism present?

25. The Platform for Privacy Preferences Project (P3P) is supposed to enable "smarter privacy tools for the web" [238]. Consider the P3P implementation outlined in the papers [185, 186].

[8]This could be viewed as a form of single sign-on.

a. Discuss the possible privacy benefits of such a system.

b. Discuss attacks on such a P3P implementation.

26. Suppose that a particular system has 1,000,000 bugs, each with MTBF of 10,000,000 hours. The good guys work for 10,000 hours and find 1,000 bugs.

    a. If Trudy works for 10 hours and finds 1 bug, what is the probability that Trudy's bug was not found by the good guys?

    b. If Trudy works for 30 hours and finds 3 bugs, what is the probability that at least one of her bugs was not found by the good guys?

27. Suppose that a large and complex piece of software has 10,000 bugs, each with an MTBF of 1,000,000 hours. Then you expect to find a particular bug after 1,000,000 hours of testing, and—since there are 10,000 bugs—you expect to find one bug for every 100 hours of testing. Suppose the good guys do 200,000 hours of testing while the bad "guy," Trudy, does 400 hours of testing.

    a. How many bugs should Trudy find? How many bugs should the good guys find?

    b. What is the probability that Trudy finds at least one bug that the good guys did not?

28. It can be shown that the probability of a security failure after t hours of testing is approximately K/t for some constant K. This implies that the mean time between failures (MTBF) is about t/K after t hours of testing. So, security improves with testing, but it only improves linearly. One implication is that to ensure an average of, say, 1,000,000 hours between security failures, we must test for (on the order of) 1,000,000 hours. Suppose that an open source software project has a MTBF of t/K. If this same project were instead closed source, we might suspect that each bug would be twice as hard for an attacker to find. If this is true, it would appear that the MTBF in the closed source case is $2t/K$ and hence the closed source project will be twice as secure for a given amount of testing t. Discuss some flaws with this reasoning.

29. This problem compares closed systems and open systems.

    a. Define "open system" and give an example of an open system.

    b. Define "closed system" and give an example of a closed system.

    c. What are the advantages of open systems, as compared to closed systems?

d. What are the advantages of closed systems, as compared to open systems?

30. Suppose that a particular open source project has MTBF $= t/K$. Without access to the source code, you believe that bugs in the software are three times as hard to find as in the open source case. If this is true, what would the MTBF be if this project were closed source?

31. Suppose that MTBF $= t^2/K$, instead of t/K. Then would there be an advantage to closed source software over open source, or vice versa, assuming that bugs are twice as hard to find in the closed source case?

32. Suppose that there are 100 security flaws in a particular software project and we can list these flaws in such a way that security flaw i requires i hours of testing to find. That is, it takes one hour to find flaw number one, two more hours to find flaw number two, three more hours to find flaw number three, and so on. What is the MTBF for this system?

33. As a deterrent to Microsoft's new Evil Death Star [210], the citizens of planet Earth have decided to build their own Good Death Star. The good citizens of Earth are debating whether to keep their Good Death Star plans secret or make the plans public.

    a. Give several reasons that tend to support keeping the plans secret.
    b. Give several reasons that tend to support making the plans public.
    c. Which case do you find more persuasive, keeping the plans secret or making the plans public? Why?

34. Suppose that you insert 100 typos into a textbook manuscript. Your editor finds 25 of these typos and, in the process, she also finds 800 other typos.

    a. Assuming that you remove all of the discovered typos and the 75 other typos that you inserted, estimate the number of typos remaining in the manuscript.
    b. What does this have to do with software security?

35. Suppose that you are asked to approximate the number of unknown bugs that remain in a particular piece of software. You insert 100 bugs into the software and then have your QA team test the software. In testing, your team discovers 40 of the bugs that you inserted, along with 120 bugs that you did not insert.

    a. Use these results to estimate the number of undiscovered bugs that remain in the program, assuming that you remove all of the discovered bugs as well as the 60 remaining bugs that you inserted.

b. Why might this test give inaccurate results?

36. Suppose that a large software company, Software Monopoly, or SM, is about to release a new software product called Doors, affectionately known as SM-Doors. The software for Doors is estimated to have 1,000,000 security flaws. It is also estimated that each security flaw that remains in the software upon release will cost SM about $20, due to lost sales resulting from damage to its reputation. SM pays its developers $100 per hour during the alpha testing phase, and at this phase, developers find flaws at a rate of about 1 flaw for every 10 hours of testing. In effect, customers act as beta testers when they find additional flaws in Doors. Suppose that SM charges $500 per copy of Doors and the estimated market for Doors is about 2,000,000 units. What is the optimal amount of alpha testing for SM to conduct?

37. Repeat Problem 36 assuming that developers find flaws at a rate of $N/100,000$ per hour of testing, where N is the number of flaws remaining in the software, and all other parameters are the same as in Problem 36. Note that this implies it is more difficult for developers to find flaws as the number of flaws decreases, which is probably more realistic than the linear assumption in Problem 36. Hint: You may want to use the fact that

$$\sum_{k=0}^{n} \frac{a}{b-k} \approx a(\ln b - \ln(b-n)).$$