Most people interact with computer programs at a surface level, entering input and eagerly (impatiently?!) awaiting a response. The public façade of most programs may be fairly thin, but most programs go much deeper than they appear at first glance. Programs have a preponderance of guts, where the real fun happens. These guts can be very complex. Exploiting software usually requires some level of understanding of software guts.

The single most important skill of a potential attacker is the ability to unravel the complexities of target software. This is called *reverse engineering* or sometimes just *reversing*. Software attackers are great tool users, but exploiting software is not magic and there are no magic software exploitation tools. To break a nontrivial target program, an attacker must manipulate the target software in unusual ways. So although an attack almost always involves tools (dissasemblers, scripting engines, input generators), these tools tend to be fairly basic. The real smarts remain the attacker's prerogative.

When attacking software, the basic idea is to grok the assumptions made by the people who created the system and then undermine those assumptions. (This is precisely why it is critical to identify as many assumptions as possible when designing and creating software.) Reverse engineering is an excellent approach to ferreting out assumptions, especially implicit assumptions that can be leveraged in an attack.[1]

---

1. A friend at Microsoft related an anecdote involving a successful attacker who made use of the word "assume" to find interesting places to attack in code. Unsuspecting developers assumed that writing about what they assumed would be OK. This is a social-level attack pattern. Similar searches through code for BUG, XXX, FIX, or TODO also tend to work.

Into the House of Logic

In some sense, programs wrap themselves around valuable data, making and enforcing rules about who can get to the data and when. The very edges of the program are exposed to the outside world just the way the interior of a house has doors at its public edges. Polite users go through these doors to get to the data they need that is stored inside. These are the entry points into software. The problem is that the very doors used by polite company to access software are also used by remote attackers.

Consider, for example, a very common kind of Internet-related software door, the TCP/IP port. Although there are many types of doors in a typical program, many attackers first look for TCP/IP ports. Finding TCP/IP ports is simple using a port-scanning tool. Ports provide public access to software programs, but finding the door is only the beginning. A typical program is complex, like a house made up of many rooms. The best treasure is usually found buried deep in the house. In all but the most trivial of exploits, an attacker must navigate complicated paths through public doors, journeying deep into the software house. An unfamiliar house is like a maze to an attacker. Successful navigation through this maze renders access to data and sometimes complete control over the software program itself.

Software is a set of instructions that determines what a general-purpose computer will do. Thus, in some sense, a software program is an instantiation of a particular machine (made up of the computer and its instructions). Machines like this obviously have explicit rules and well-defined behavior. Although we can watch this behavior unfold as we run a program on a machine, looking at the code and coming to an understanding of the inner workings of a program sometimes takes more effort. In some cases the source code for a program is available for us to examine; other times, it is not. Therefore, attack techniques must not always rely on having source code. In fact, some attack techniques are valuable regardless of the availability of source code. Other techniques can actually reconstruct the source code from the machine instructions. These techniques are the focus of this chapter.

Reverse Engineering

Reverse engineering is the process of creating a blueprint of a machine to discern its rules *by looking only at the machine and its behavior*. At a high level, this process involves taking something that you may not completely understand technically when you start, and coming to understand completely

its function, its internals, and its construction. A good reverse engineer attempts to understand the details of software, which by necessity involves understanding how the overall computing machinery that the software runs on functions. A reverse engineer requires a deep understanding of both the hardware *and* the software, and how it all works together.

Think about how external input is handled by a software program. External "user" input can contain commands and data. Each code path in the target involves a number of control decisions that are made based on input. Sometimes a code path will be wide and will allow any number of messages to pass through successfully. Other times a code path will be narrow, closing things down or even halting if the input isn't formatted exactly the right way. This series of twists and turns can be mapped if you have the right tools. Figure 3–1 illustrates code paths as found in a common FTP server program. In this diagram, a complex subroutine is being mapped. Each location is shown in a box along with the corresponding machine instructions.

Generally speaking, the deeper you go as you wander into a program, the longer the code path between the input where you "start" and the place where you end up. Getting to a particular location in this house of logic requires following paths to various rooms (hopefully where the valuables are). Each internal door through which you pass imposes rules on the kinds of messages that may pass. Wandering from room to room thus involves negotiating multiple sets of rules regarding the input that will be accepted. This makes crafting an input stream that can pass through lots of doors (both external and internal) a real challenge. In general, attack input becomes progressively more refined and specific as it digs deeper into a target program. This is precisely why attacking software requires much more than a simple brute-force approach. Simply blasting a program with random input almost never traverses all the code paths. Thus, many possible paths through the house remain unexplored (and unexploited) by both attackers and defenders.

Why Reverse Engineer?

Reverse engineering allows you to learn about a program's structure and its logic. Reverse engineering thus leads to critical insights regarding how a program functions. This kind of insight is extremely useful when you exploit software. There are obvious advantages to be had from reverse engineering. For example, you can learn the kind of system functions a target program is using. You can learn the files the target program accesses. You

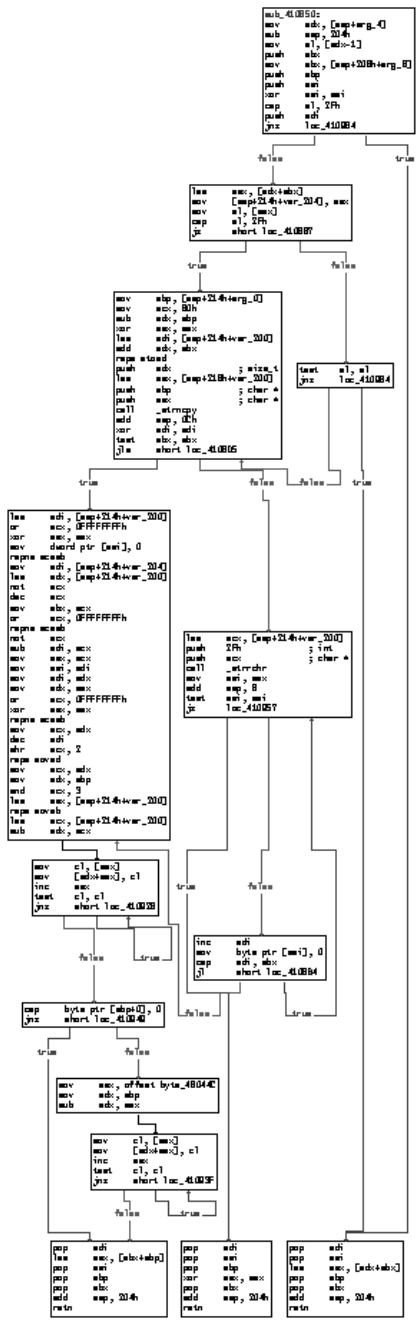


Figure 3–1 This graph illustrates control flow through a subroutine in a common FTP server. Each block is a set of instructions that runs as a group, one instruction after the other. The lines between boxes illustrate the ways that control in the code connects boxes. There are various "branches" between the boxes that represent decision points in the control flow. In many cases, a decision regarding how to branch can be influenced by data supplied by an attacker.

can learn the protocols the target software uses and how it communicates with other parts of the target network.

The most powerful advantage to reversing is that you can change a program's structure and thus directly affect its logical flow. Technically this activity is called *patching,* because it involves placing new code patches (in a seamless manner) over the original code, much like a patch stitched on a blanket. Patching allows you to add commands or change the way particular function calls work. This enables you to add secret features, remove or disable functions, and fix security bugs without source code. A common use of patching in the computer underground involves removing copy protection mechanisms.

Like any skill, reverse engineering can be used for good and for bad ends.

Should Reverse Engineering Be Illegal?

Because reverse engineering can be used to reconstruct source code, it walks a fine line in intellectual property law. Many software license agreements strictly forbid reverse engineering. Software companies fear (and rightly so) that their trade secret algorithms and methods will be more directly revealed through reverse engineering than they are through external machine observation. However, there is no general-purpose law against reverse engineering.

Because reverse engineering is a crucial step in removing copy protection schemes that are often used in the computer underground, there is some confusion regarding its legality. Patching software to defeat copy protection or digital rights management schemes is illegal. Reverse engineering software is not. If the law changes and reverse engineering is made illegal, then a serious blow will be dealt to the common user of software (especially the common and curious user). A law completely outlawing reverse engineering would be like a law making it illegal to open the hood of your car to repair it. Under such a system, car users would be required by law to go to the dealership for all repairs and maintenance.[2]

Software vendors forbid reverse engineering in their license agreements for many reasons. One reason is that reverse engineering does, in fact, more obviously reveal secret methods. But all this is a bit silly, really. To a skilled

2. Although this may not sound so bad to you, note that such a law may well make it illegal for any "nonauthorized" mechanic to work on your car as well.

reverse engineer, looking at the binary machine code of a program is just as good as having the source code. So the secret is already out, but in this case only specialists can "read" the code. Note that secret methods can be defended through means other than attempting to hide them from everyone but specialists in compiled code. Patents exist specifically for this purpose, and so does copyright law. A good example of properly protecting a program can be found in the data encryption algorithms domain. To be acceptable as actually useful and powerful, encryption algorithms must be published for the cryptographic world to evaluate. However, the inventor of the algorithm can maintain rights to the work. Such was the case with the popular RSA encryption scheme. Also note that although this book is copyrighted, you are allowed to read it and understand it. In fact, you're encouraged to do so.

Another reason that software vendors would like to see reverse engineering made illegal is to prevent researchers from finding security flaws in their code. Quite often security researchers find flaws in software and report them in public forums like bugtraq. This makes software vendors look bad, hurts their image, and damages their reputation as upstanding software vendors. (It also tends to make software improve at the same time.) A well-established practice is for a security specialist to report a flaw to the vendor and give them a reasonable grace period to fix the bug before its existence is made public. Note that during this grace period the flaw still exists for more secretive security specialists (including bad guys) to exploit. If reverse engineering is made illegal, then researchers will be prevented from using a critical tool for evaluating the quality of code. Without the ability to examine the structure of software, users will be forced to take the vendor's word that the software is truly a quality product.[3] Keep in mind that no vendor is currently held financially liable for failures in its software. We can thus trust the vendor's word regarding quality as far as it impacts their bottom line (and no farther).

The Digital Millennium Copyright Act (DMCA) explicitly (and controversially) addresses reverse engineering from the perspective of copyright infringement and software cracking. For an interesting view of how this law impacts individual liberty, check out Ed Felten's Web site at http://www.freedomtotinker.com.

---

3. Note that many consumers already know that they are being sold poor-quality software, but some consumers remain confused about how much quality can actually be attained in software.

When you purchase or install software, you are typically presented with an end-user license agreement (EULA) on a click-through screen. This is a legal agreement that you are asked to read and agree to. In many cases, simply physically opening a software package container, such as the box or the disk envelope, implies that you have agreed to the software license. When you download software on-line, you are typically asked to press "I AGREE" in response to a EULA document displayed on the Web site (we won't get into the security ramifications of this). These agreements usually contain language that strictly prohibits reverse engineering. However, these agreements may or may not hold up in court [Kaner and Pels, 1998].

The Uniform Computer Information Transactions Act (UCITA) poses strong restrictions on reverse engineering and may be used to help "click through" EULA's stand-up in court. Some states have adopted the UCITA (Maryland and Virginia as of this writing), which strongly affects your ability to reverse engineer legally.

Reverse Engineering Tools and Concepts

Reverse engineering fuels entire technical industries and paves the way for competition. Reverse engineers work on hard problems like integrating software with proprietary protocols and code. They also are often tasked with unraveling the mysteries of new products released by competitors. The boom in the 1980s of the PC clone market was heavily driven by the ability to reverse engineer the IBM PC BIOS software. The same tricks have being applied in the set-top game console industry (which includes the Sony PlayStation, for example). Chip manufacturers Cyrix and AMD have reverse engineered the Intel microprocessor to release compatible chips. From a legal perspective, reverse engineering work is dangerous because it skirts the edges of the law. New laws such as the DMCA and UCITA (which many security analysts decry as egregious), put heavy restrictions on reverse engineering. If you are tasked with reverse engineering software legally, you need to understand these laws. We are not going to dwell on the legal aspects of reverse engineering because we are not legal experts. Suffice it to say that it is very important to seek legal counsel on these matters, especially if you represent a company that cares about its intellectual property.

The Debugger

A debugger is a software program that attaches to and controls other software programs. A debugger allows single stepping of code, debug tracing,

setting breakpoints, and viewing variables and memory state in the target program as it executes in a stepwise fashion. Debuggers are invaluable in determining logical program flow. Debuggers fall into two categories: user-mode and kernel-mode debuggers. User-mode debuggers run like normal programs under the OS and are subject to the same rules as normal programs. Thus, user-mode debuggers can only debug other user-level processes. A kernel-mode debugger is part of the OS and can debug device drivers and even the OS itself. One of the most popular commercial kernel-mode debuggers is called SoftIce and it is published by Compuware (http://www.compuware.com/products/driverstudio/ds/softice.htm).

Fault Injection Tools

Tools that can supply malformed or improperly formatted input to a target software process to cause failures are one class of fault injection tool. Program failures can be analyzed to determine whether errors exist in the target software. Some failures have security implications, such as failures that allow an attacker direct access to the host computer or network. Fault injection tools fall into two categories: host and network. Host-based fault injectors operate like debuggers and can attach to a process and alter program states. Network-based fault injectors manipulate network traffic to determine the effect on the receiver.

Although classic approaches to fault injection often make use of source code instrumentation [Voas and McGraw, 1999], some modern fault injectors pay more attention to tweaking program input. Of particular interest to security practitioners are Hailstorm (Cenzic), the Failure Simulation Tool or FST (Cigital), and Holodeck (Florida Tech). James Whittaker's approach to fault injection for testing (and breaking) software is explained in two books [Whittaker, 2002; Whittaker and Thompson, 2003].

The Disassembler

A disassembler is a tool that converts machine-readable code into assembly language. Assembly language is a human-readable form of machine code (well, more human readable than a string of bits anyway). Disassemblers reveal which machine instructions are being used in the code. Machine code is usually specific to a given hardware architecture (such as the PowerPC chip or Intel Pentium chip). Thus, disassemblers are written expressly for the target hardware architecture.

The Reverse Compiler or Decompiler

A decompiler is a tool that converts assembly code or machine code into source code in a higher level language such as C. Decompilers also exist to transform intermediate languages such as Java byte code and Microsoft Common Runtime Language (CRL) into source code such as Java. These tools are extremely helpful in determining higher level logic such as loops, switches, and if-then statements. Decompilers are much like disassemblers but take the process one (important) step further. A good disassembler/compiler pair can be used to compile its own collective output back into the same binary.

Approaches to Reverse Engineering

As we said earlier, sometimes source code is available for a reverse engineer and sometimes it is not. White box and black box testing and analysis methods both attempt to understand the software, but they use different approaches depending on whether the analyst has access to source code.

Regardless of the method, there are several key areas that an attacker should examine to find vulnerabilities in software:

- Functions that do improper (or no) bounds checking
- Functions that pass through or consume user-supplied data in a format string
- Functions meant to enforce bounds checking in a format string (such as `%20s`)
- Routines that get user input using a loop
- Low-level byte copy operations
- Routines that use pointer arithmetic on user-supplied buffers
- "Trusted" system calls that take dynamic input

This somewhat tactical list is useful when you are "in the weeds" with binary code.

White Box Analysis

White box analysis involves analyzing and understanding source code. Sometimes only binary code is available, but if you decompile a binary to get source code and then study the code, this can be considered a kind of white box analysis as well. White box testing is typically very effective in

finding programming errors and implementation errors in software. In some cases this activity amounts to pattern matching and can even be automated with a static analyzer.[4] One drawback to this kind of whitebox testing is that it may report a potential vulnerability where none actually exists (called a *false positive*). Nevertheless, using static analysis methods on source code is a good approach to exploiting some kinds of software.

There are two types of white box analysis tools, those that require source code and those that automatically decompile the binary code and continue from there. One powerful and commercially available white box analysis platform, called IDA-Pro, does not require source code access. SourceScope, which includes an extensive database of source code-related problems and issues commonly encountered in Java, C, and C++, does require source code. The knowledge encapsulated in these tools is extremely useful in security analysis (and, of course, in exploiting software).

Black Box Analysis

Black box analysis refers to analyzing a running program by probing it with various inputs. This kind of testing requires only a running program and does not make use of source code analysis of any kind. In the security paradigm, malicious input can be supplied to the program in an effort to cause it to break. If the program does break during a particular test, then a security problem may have been discovered.

Note that black box testing is possible even without access to binary code. That is, a program can be tested remotely over a network. All that is required is a program running somewhere that is accepting input. If the tester can supply input that the program consumes (and can observe the effect of the test), then black box testing is possible. This is one reason that real attackers often resort to black box techniques.

Black box testing is not as effective as white box testing in obtaining knowledge of the code and its behavior, but black box testing is much easier to accomplish and usually requires much less expertise than white box testing. During black box testing, an analyst attempts to evaluate as many meaningful internal code paths as can be directly influenced and observed from outside the system. Black box testing cannot exhaustively search a real program's input space for problems because of theoretical

4. Cigital's tool SourceScope, for example, can be used to find potential security flaws in a piece of software given its source code (http://www.cigital.com).

constraints, but a black box test does act more like an actual attack on target software in a real operational environment than a white box test usually can.

Because black box testing happens on a live system, it is often an effective way of understanding and evaluating denial-of-service problems. And because black box testing can validate an application *within its runtime environment* (if possible), it can be used to determine whether a potential problem area is actually vulnerable in a real production system.[5] Sometimes problems that are discovered in a white box analysis may not be exploitable in a real, deployed system. A firewall may block the attack, for example.[6]

Cenzic's Hailstorm is a commercially available black box testing platform for networked software. It can be used to probe live systems for security problems. For testing network routers and switches, special hardware devices are available, such as SmartBits and IXIA. A freeware tool called ISICS can be used to probe TCP/IP stack integrity. Protocol attack systems that use black box techniques include PROTOS and Spike.

Gray Box Analysis

Gray box analysis combines white box techniques with black box input testing. Gray box approaches usually require using several tools together. A good example of a simple gray box analysis is running a target program within a debugger and then supplying particular sets of inputs to the program. In this way, the program is exercised while the debugger is used to detect any failures or faulty behavior. Rational's Purify is a commercial tool that can provide detailed runtime analysis focused on memory use and consumption. This is particularly important for C and C++ programs (in which memory problems are rampant). A freeware debugger that provides runtime analysis for Linux is called Valgrind.

All testing methods can reveal possible software risks and potential exploits. White box analysis directly identifies more bugs, but the actual risk of exploit is hard to measure. Black box analysis identifies real problems

5. The problem with testing on live production systems should be obvious. A successful denial-of-service test will take down a production system just as effectively as a real attack. Companies are not very receptive to this sort of testing, in our experience.
6. However, note that white box analysis is useful for testing how a piece of software will behave across multiple environments. For code that is widely deployed, this kind of testing is essential.

that are known to be exploitable. The use of gray box techniques combines both methods in a powerful way. Black box tests can scan programs across networks. White box tests require source code or binaries to analyze statically. In a typical case, white box analysis is used to find potential problem areas, and black box testing is then used to develop working attacks against these areas.

Black Box	*White Box*
audit software runtime environment External threats Denial of service Cascade failure Security policy and filters Scales and runs across enterprise network Valuable to security/systems administrators	Audit software code Programming errors Central code repository required Valuable to developers and testers

One problem with almost all kinds of security testing (regardless of whether such testing is black box or white box) is that there really isn't any. That is, most QA organizations concern themselves with functional testing and spend very little time understanding or probing for security risks. The QA process is almost always broken in most commercial software houses anyway because of time and budget constraints and the belief that QA is not an essential part of software development.

As software becomes more important, more emphasis is being placed on software quality management—a unified approach to testing and analysis that encompasses security, reliability, and performance. Software quality management uses both white box and black box techniques to identify and manage software risks as early as possible in the software development life cycle.

Using Gray Box Techniques to Find Vulnerabilities in Microsoft SQL Server 7

Gray box techniques usually leverage several tools. We provide an example using runtime debugging tools combined with a black box input generator.

Using runtime error detection and debugging tools is a powerful way of finding problem software. When combined with black box injection tools, debuggers help catch software faults. In many cases, disassembly of the program can determine the exact nature of a software bug like the one we will show you.

One very powerful tool that examines software dynamically as it runs is Rational's Purify. In this example, we perform black box injection against Microsoft's SQL Server 7 using Hailstorm, while monitoring the target instrumented under Purify. By combining Purify and Hailstorm, the test is able to uncover a memory corruption problem occurring in the SQL server as a result of malformed protocol input. The corruption results in a software exception and subsequent failure.

To start, a remote input point is identified in the SQL server. The server listens for connections on TCP port 1433. The protocol used over this port is undocumented for the most part. Instead of reverse engineering the protocol, a simple test is constructed that supplies random inputs interspersed with numerical sequences. These data are played against the TCP port. The result is the generation of many possible "quasilegal" inputs to the port, which thus covers a wide range of input values. The inputs are injected for several minutes at a rate of around 20 per second.

The data injected pass through a number of different code paths inside the SQL server software. These locations, in essence, read the protocol header. After a short time, the test causes a fault, and Purify notes that memory corruption has occurred.

The screen shot in Figure 3–2 illustrates the SQL server failure, the Purify dump, and the Hailstorm testing platform all in one place. The memory corruption noted by Purify occurs before the SQL server crashes. Although the attack does result in a server crash, the point of memory corruption would be hard to determine without the use of Purify. The data supplied by Purify allow us to locate the exact code path that failed.

The detection of this failure occurs well before an actual exploit has occurred. If we wanted to find this exploit using only black box tools, we might spend days trying input tests before this bug is exercised. The corruption that is occurring might cause a crash in an entirely different code location, making it very hard to identify which input sequence causes the error. Static analysis might have detected a memory corruption problem, but it would never be able to determine whether the bug could be exploited in practice by an attacker. By combining both technologies as we do in this example, we save time and get the best of both worlds.

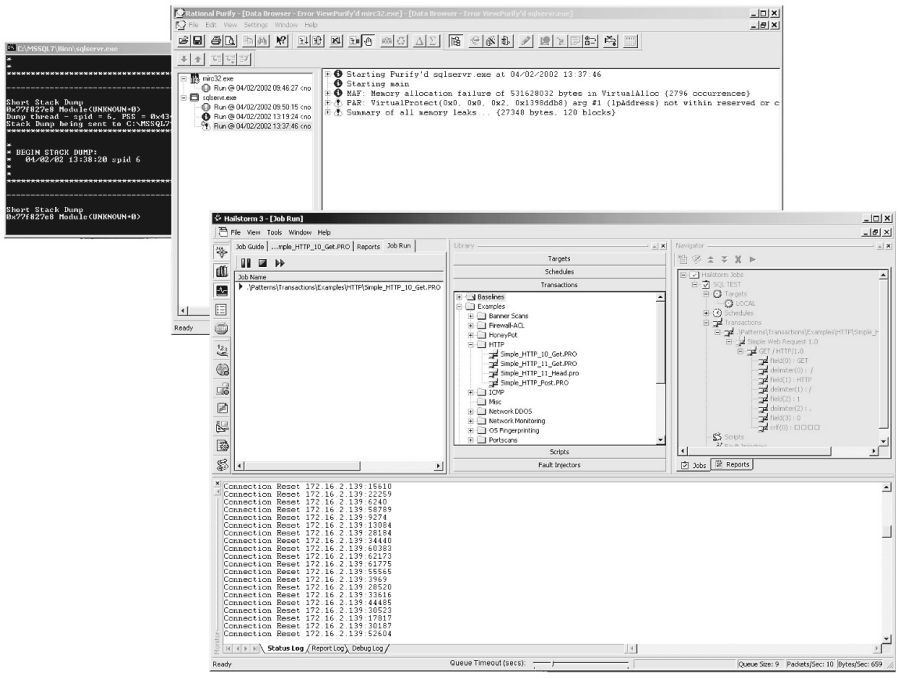


Figure 3–2 Screen shots of Hailstorm and Purify being used to probe the SQL server software for security problems using a black box paradigm.

Methods of the Reverser

There are several methods that can be used while reverse engineering software. Each has benefits and each has resource and time requirements. A typical approach uses a mixture of methods when decompiling and examining software. The best method mix depends entirely on your goals. For example, you may first want to run a quick scan of the code for obvious vulnerabilities. Next, you may want to perform a detailed input trace on the user-supplied data. You may not have time to trace each and every path, so you may use complex breakpoints and other tools to speed up the process. What follows is a brief description of several basic methods.

Tracing Input

Input tracing is the most thorough of all methods. First you identify the input points in the code. Input points are places where user-supplied data are being delivered to the program. For example, a call to `WSARecvFrom( )` will retrieve a network packet. This call, in essence, accepts user-supplied data from the network and places it in a buffer. You can set a breakpoint on

the input point and single-step trace into the program. Of course, your debugging tools should always include a pencil and paper. You must note each twist and turn in the code path. This approach is very tedious, but it is also very comprehensive.

Although determining all input points takes a great deal of time if you do it by hand, you have the opportunity to note every single code location that makes decisions based on user-supplied data. Using this method you can find very complex problems.

One language that protects against this kind of "look through the inputs" attack is Perl. Perl has a special security mode called *taint mode*. Taint mode uses a combination of static and dynamic checks to monitor all information that comes from outside a program (such as user input, program arguments, and environment variables) and issues warnings when the program attempts to do something potentially dangerous with that untrusted information. Consider the following script:

```
#!/usr/bin/perl -T
$username = <STDIN>;
chop $username;
system ("cat /usr/stats/$username");
```

On executing this script, Perl enters taint mode because of the `-T` option passed in the invocation line at the top. Perl then tries to compile the program. Taint mode will notice that the programmer has not explicitly initialized the `PATH` variable, yet tries to invoke a program using the shell anyway, which can easily be exploited. It issues an error such as the following before aborting compilation:

```
Insecure $ENV{PATH} while running with -T switch at
./catform.pl line 4, <STDIN> chunk 1.
```

We can modify the script to set the program's path explicitly to some safe value at startup:

```
#!/usr/bin/perl -T
use strict;
$ENV{PATH} = join ':' => split (" ",<< '__EOPATH__');
 /usr/bin
 /bin
__EOPATH__
my $username = <STDIN>;
```

```
chop $username;
system ("cat /usr/stats/$username");
```

Taint mode now determines that the `$username` variable is externally controlled and is not to be trusted. It determines that, because `$username` may be poisoned, the call to `system` may be poisoned. It thus gives another error:

```
Insecure dependency in system while running with
-T switch at ./catform.pl line 9, <STDIN> chunk 1.
```

Even if we were to copy `$username` into another variable, taint mode would still catch the problem.

In the previous example, taint mode complains because the variable can use shell magic to cause a command to run. But taint mode does not address every possible input vulnerability, so a clever attacker using our input-driven method can still win.

Advanced dataflow analysis is also useful to help protect against our attack method (or to help carry it out). Static analysis tools can help an analyst (or an attacker) identify all possible input points and to determine which variables are affected from the outside. The security research literature is filled with references discussing "secure information flow" that take advantage of data flow analysis to determine program safety.

Exploiting Version Differences

When you study a system to find weaknesses, remember that the software vendor fixes many bugs in each version release. In some cases the vendor may supply a "hot fix" or a patch that updates the system binaries. It is extremely important to watch the differences between software versions.

The differences between versions are, in essence, attack maps. If a new version of the software or protocol specification is available, then weaknesses or bugs will most certainly have been fixed (if they have been discovered). Even if the "bug fix" list is not published, you can compare the binary files of the older version against the new. Differences can be uncovered where features have been added or bugs have been fixed. These differences thereby reveal important hints regarding where to look for vulnerabilities.

Making Use of Code Coverage

Cracking a computer system is a scientific process just as much as it is an art. In fact, wielding the scientific method gives the attacker an upper hand in an otherwise arbitrary game. The scientific method starts with measurement. Without the ability to measure your environment, how can you possibly draw conclusions about it? Most of the approaches we consider in this text are designed to find programming flaws. Usually (not always), the bugs we find this way are confined to small regions of code. In other words, it's usually the small coding mistakes that we are after. This is one reason that new development tools are very likely to hamper many of the traditional methods of attack. It's easy for a development tool to identify a simple programming error (statically) and compile it out. In a few years, buffer overflows will be obsolete as an attack method.

All the techniques we describe are a form of measurement. We observe the behavior of the program while it is exercised in some way (for example, placed under stress). Strange behavior usually indicates unstable code. Unstable code has a high probability of security weaknesses. Measurement is the key.

Code coverage is an important type of measurement—perhaps the most important. Code coverage is a way of watching a program execute and determining which code paths have been exercised. Many tools are available for code coverage analysis. Code coverage tools do not always require source code. Some tools can attach to a process and gather measurements in real time. For one example, check out the University of Maryland's tool dyninstAPI (created by Jeff Hollingsworth).[7]

As an attacker, code coverage tells you how much work is left to do when you're surveying the landscape. By using coverage analysis you can immediately learn what you have missed. Computer programs are complex, and cracking them is tedious business. It's human nature to skip parts of the code and take shortcuts. Code coverage can show you whether you have missed something. If you skipped that subroutine because it looked harmless, well think again! Code coverage can help you go back and check your work, walking down those dark alleys you missed the first time.

If you are trying to crack software, you most likely start with the user

7. The dyninstAPI tool can be found at http://www.dyninst.org/.

input point. As an example, consider a call to `WSARecv()`.[8] Using outside-in tracing, you can measure the code paths that are visited. Many decisions are made by the code after user input is accepted. These decisions are implemented as branching statements, such as the conditional branch statements JNZ and JE, in x86 machine code. A code coverage tool can detect when a branch is about to occur and can build a map of each continuous block of machine code. What this means is that you, as the attacker, can instantly determine which code paths you have not exercised during your analysis.

Reverse engineers know that their work is long and tedious. Using code coverage gives the clever reverse engineer a map for tracking progress. Such tracking can keep you sane and can also keep you going when you otherwise might give up without exploring all opportunities.

Code coverage is such an important tool for your bag of tricks that later in the chapter we illustrate how you can build a code coverage tool from scratch. In our example we focus on the x86 assembly language and the Windows XP OS. Our experience leads us to believe that it will be hard for you to find the perfect code coverage tool for your exact needs. Many of the available tools, commercial or otherwise, lack attack-style features and data visualization methods that are important to the attacker.

Accessing the Kernel

Poor access controls on handles opened by drivers can expose a system to attack. If you find a device driver with an unprotected handle, you might be able to run `IOCTL` commands to the kernel driver. Depending on what the driver supports, you might be able to crash the machine or gain access to the kernel. Any input to the driver that includes memory addresses should be immediately tested by inserting `NULL` values. Another option is to insert addresses that map to kernel memory. If the driver doesn't perform sanity checking on the user-mode-supplied values, kernel memory may get malformed. If the attack is very clever, global state in the kernel may be modified, altering access permissions.

Leaking Data in Shared Buffers

Sharing buffers is somewhat like sharing food. A restaurant (hopefully) maintains strict rules about where raw meat can be placed. A little raw

8. The `WSARecv` function receives data from a connected socket. See http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winsock/winsock/wsarecv_2.asp.

juice in someone's cooked meal could lead to illness and a lawsuit. A typical program has many buffers. Programs tend to reuse the same buffers over and over, but the questions from our perspective are the following: Will they be cleaned? Are dirty data kept from clean data? Buffers are a great place to start looking for potential data leakage. Any buffer that is used for both public and private data has a potential to leak information.

Attacks that cause state corruption and/or race conditions may be used to cause private data to leak into public data. Any use of a buffer without cleaning the data between uses leads to potential leaks.

Example: The Ethernet Scrubbing Problem

One of us (Hoglund) codiscovered a vulnerability a few years ago that affects potentially millions of ethernet cards worldwide.[9] Ethernet cards use standard chip sets to connect to the network. These chips are truly the "tires" of the Internet. The problem is that many of these chips are leaking data across packets.

The problem exists because data are stored in a buffer on the ethernet microchip. The minimum amount of data that must be sent in an ethernet packet is 66 bytes. This is the minimum frame size. But, many packets that need to be transmitted are actually much smaller than 66 bytes. Examples include small ping packets and ARP requests. Thus, these small packets are padded with data to meet the minimum number of 66 bytes.

The problem? Many chips do not clean their buffers between packets. Thus, a small packet will be padded with whatever was left in the buffer from the last packet. This means that other people's packets are leaking into a potential attack packet. This attack is simple to exploit and the attack works over switched environments. An attack can craft a volley of small packets that solicit a small packet as a reply. As the small reply packets arrive, the attacker looks at the padding data to see other people's packet data.

Of course, some data are lost in this attack, because the first part of every packet is overwritten with the legitimate data for the reply. So, the attacker will naturally want to craft as small a packet as possible to siphon the data stream. Ping packets work well for these purposes, and allow an attacker to sniff cleartext passwords and even parts of encryption keys. ARP packets are even smaller, but will not work as a remote attack. Using ARP

9. This vulnerability was later released independently as the "Etherleak vulnerability." Go to http://archives.neohapsis.com/archives/vulnwatch/2003-q1/0016.html for more information.

packets, an attacker can get TCP ACK numbers from other sessions in the response. This aids in a standard TCP/IP hijacking attack.[10]

Auditing for Access Requirement Screwups

Lack of planning or laziness on the part of software engineers often leads to programs that require administrator or root access to operate.[11] Many programs that were upgraded from older Windows environments to work on Win2K and Windows XP usually require full access to the system. This would be OK except that programs that operate this way tend to leave a lot of world-accessible files sitting around.

Look for directories where user data files are being stored. Ask yourself, are these directories storing sensitive data as well? If so, is the directory permission weak? This applies to the NT registry and to database operations as well. If an attacker replaces a DLL or changes the settings for a program, the attacker might be able to elevate access and take over a system. Under Windows NT, look for open calls that request or create resources with no access restrictions. Excessive access requirements lead to insecure file and object permissions.

Using Your API Resources

Many system calls are known to lead to potential vulnerabilities [Viega and McGraw, 2001]. One good method of attack when reversing is to look for known calls that are problematic (including, for example, the much maligned `strcpy()`). Fortunately, there are tools that can help.[12]

Figure 3–3 includes a screenshot that shows APISPY32 capturing all calls to `strcpy` on a target system. We used the APISPY32 tool to capture a series of `lstrcpy` calls from Microsoft SQL server. Not all calls to `strcpy` are going to be vulnerable to buffer overflow, but some will.

APISPY is very easy to set up. You can download the program from www.internals.com. You must make a special file called APISpy32.api and place it in the WINNT or WINDOWS directory. For this example, we use the following configuration file settings:

---

10. See *Firewalls and Internet Security* [Cheswick et al., 2003] for more on TCP/IP hijacking.
11. To learn more about this common problem and how to avoid it, see *Building Secure Software* [Viega and McGraw, 2001].
12. Cigital maintains a database of static analysis rules pertaining to security. There are more than 550 entries for C and C++ alone. Static analysis tools use this information to uncover potential vulnerabilities in software (an approach that works as well for software exploit as it does for software improvement).

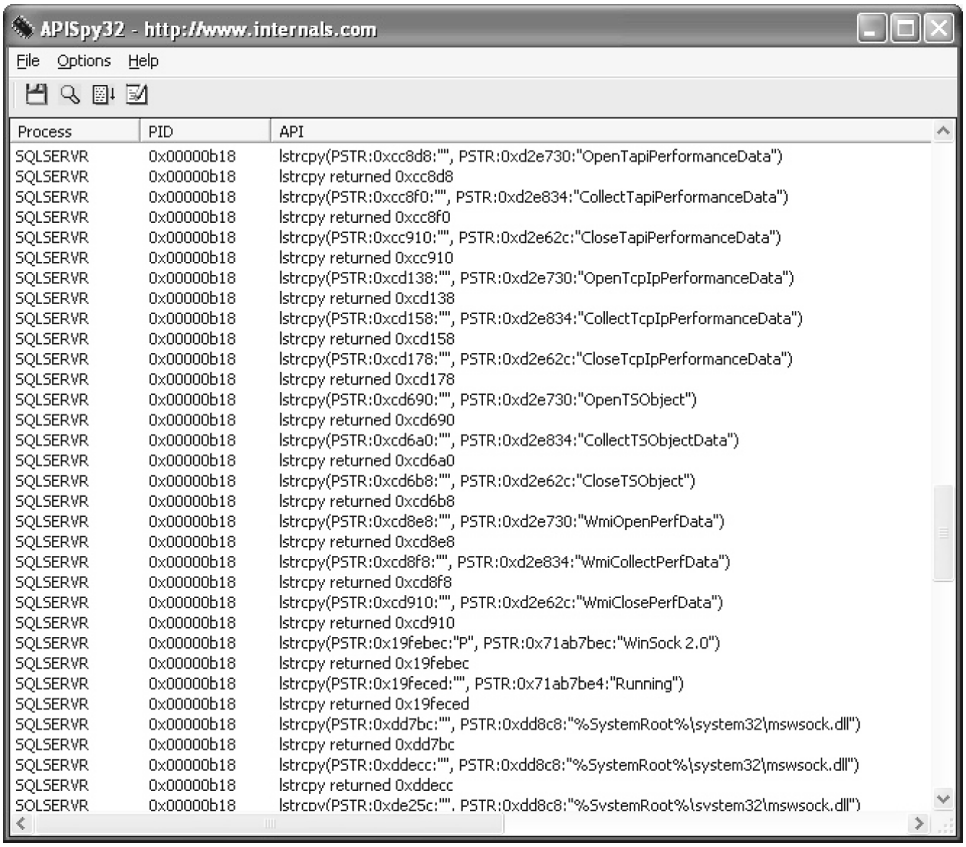


Figure 3–3 APISPY32 can be used to find `lstrcpy()` calls in the SQL server code. This screenshot shows the results of one query.

```
KERNEL32.DLL:lstrcpy(PSTR, PSTR)
KERNEL32.DLL:lstrcpyA(PSTR, PSTR)
KERNEL32.DLL:lstrcat(PSTR, PSTR)
KERNEL32.DLL:lstrcatA(PSTR, PSTR)
WSOCK32.DLL:recv
WS2_32.DLL:recv
ADVAPI32.DLL:SetSecurityDescriptorDACL(DWORD, DWORD, DWORD, DWORD)
```

This sets APISPY to look for some function calls that we are interested in. While testing, it is extremely useful to hook potentially vulnerable API calls, as well as any calls that take user input. In between the two comes your reverse engineering task. If you can determine that data from the input side reaches the vulnerable API call, you have found yourself a way in.

Writing Interactive Disassembler (IDA) Plugins

IDA is short for Interactive Disassembler (available from www.datarescue.com) and is one of the most popular reverse engineering tools for software. IDA supports plugin modules so customers can extend the functionality and automate tasks. For this book we created a simple IDA plugin that can scan through two binary files and compare them. The plugin will highlight any code regions that have changed. This can be used to compare a prepatch executable with a postpatch executable to determine which lines of code were fixed.

In many cases, software vendors will "secretly" fix security bugs. The tool we provide here can help an attacker find these secret patches. Be forewarned that this plugin can flag many locations that have not changed at all. If compiler options are changed or the padding between functions is altered, the plugin will return a nice set of false positives. Nonetheless, this is a great example to illustrate how to start writing IDA plugins.

Our example also emphasizes the biggest problem with penetrate-and-patch security. Patches are really attack maps, and clever attackers know how to read them. To use this code you will need the IDA software development kit (SDK), which is available along with the IDA product. Code is commented inline. These are standard header files. Depending on which API calls you intend to use, you may need to include other header files. Note that we have disabled a certain warning message and included the Windows header file as well. By doing this we are able to use Windows graphical user interface (GUI) code for pop-up dialogs and so on. The warning 4273 is thrown when you use the standard template library and it's customary to disable it.

```
#include <windows.h>
#pragma warning( disable:4273 )
#include <ida.hpp>
#include <idp.hpp>
#include <bytes.hpp>
#include <loader.hpp>
#include <kernwin.hpp>
#include <name.hpp>
```

Because our plugin is based on a sample plugin supplied with the SDK, the following code is merely part of the sample. These are required functions and the comments were already part of the sample.

```
//--------------------------------------------------------------------------
// This callback is called for UI notification events.
static int sample_callback(void * /*user_data*/, int event_id, va_list /*va*/)
{
 if ( event_id != ui_msg )   // Avoid recursion.
  if ( event_id != ui_setstate
   && event_id != ui_showauto
   && event_id != ui_refreshmarked ) // Ignore uninteresting events
         msg("ui_callback %d\n", event_id);
 return 0;           // 0 means "process the event";
                     // otherwise, the event would be ignored.
}
//--------------------------------------------------------------------------
// A sample of how to generate user-defined line prefixes
static const int prefix_width = 8;

static void get_user_defined_prefix(ea_t ea,
                                    int lnnum,
                                    int indent,
                                    const char *line,
                                    char *buf,
                                    size_t bufsize)
{
 buf[0] = '\0';    // Empty prefix by default

 // We want to display the prefix only on the lines which
 // contain the instruction itself.

 if ( indent != -1 ) return;       // A directive
 if ( line[0] == '\0' ) return;    // Empty line
 if ( *line == COLOR_ON ) line += 2;
 if ( *line == ash.cmnt[0] ) return;  // Comment line. . .

 // We don't want the prefix to be printed again for other lines of the
 // same instruction/data. For that we remember the line number
 // and compare it before generating the prefix.

 static ea_t old_ea = BADADDR;
 static int old_lnnum;
 if ( old_ea == ea && old_lnnum == lnnum ) return;

 // Let's display the size of the current item as the user-defined prefix.
 ulong our_size = get_item_size(ea);
```

```
 // Seems to be an instruction line. We don't bother with the width
 // because it will be padded with spaces by the kernel.

 snprintf(buf, bufsize, " %d", our_size);
 // Remember the address and line number we produced the line prefix for.
 old_ea = ea;
 old_lnnum = lnnum;

}

//--------------------------------------------------------------------------
//
//   Initialize.
//
//   IDA will call this function only once.
//   If this function returns PLGUIN_SKIP, IDA will never load it again.
//   If this function returns PLUGIN_OK, IDA will unload the plugin but
//   remember that the plugin agreed to work with the database.
//   The plugin will be loaded again if the user invokes it by
//   pressing the hot key or by selecting it from the menu.
//   After the second load, the plugin will stay in memory.
//   If this function returns PLUGIN_KEEP, IDA will keep the plugin
//   in memory. In this case the initialization function can hook
//   into the processor module and user interface notification points.
//   See the hook_to_notification_point() function.
//
//   In this example we check the input file format and make the decision.
//   You may or may not check any other conditions to decide what you do,
//   whether you agree to work with the database.
//
int init(void)
{
 if ( inf.filetype == f_ELF ) return PLUGIN_SKIP;

// Please uncomment the following line to see how the notification works:
// hook_to_notification_point(HT_UI, sample_callback, NULL);

// Please uncomment the following line to see how the user-defined prefix works:
// set_user_defined_prefix(prefix_width, get_user_defined_prefix);
 return PLUGIN_KEEP;
}

//--------------------------------------------------------------------------
//   Terminate.
//   Usually this callback is empty.
```

```
//   The plugin should unhook from the notification lists if
//   hook_to_notification_point() was used.
//
//   IDA will call this function when the user asks to exit.
//   This function won't be called in the case of emergency exits.

void term(void)
{
 unhook_from_notification_point(HT_UI, sample_callback);
 set_user_defined_prefix(0, NULL);
}
```

A few more header files and some global variables are included here:

```
#include <process.h>
#include "resource.h"

DWORD g_tempest_state = 0;
LPVOID g_mapped_file = NULL;
DWORD g_file_size = 0;
```

This function loads a file into memory. This file is going to be used as the target to compare our loaded binary against. Typically you would load the unpatched file into IDA and compare it with the patched file:

```
bool load_file( char *theFilename )
{
    HANDLE aFileH =
        CreateFile(    theFilename,
                               GENERIC_READ,
                               0,
                               NULL,
                               OPEN_EXISTING,
                               FILE_ATTRIBUTE_NORMAL,
                               NULL);

    if(INVALID_HANDLE_VALUE == aFileH)
    {
        msg("Failed to open file.\n");
        return FALSE;
    }

    HANDLE aMapH =
        CreateFileMapping(    aFileH,
```

```
                                        NULL,
                                        PAGE_READONLY,
                                        0,
                                        0,
                                        NULL );
    if(!aMapH)
    {
        msg("failed to open map of file\n");
        return FALSE;
    }


    LPVOID aFilePointer =
        MapViewOfFileEx(
                aMapH,
                FILE_MAP_READ,
                0,
                0,
                0,
                NULL);

    DWORD aFileSize = GetFileSize(aFileH, NULL);

    g_file_size = aFileSize;
    g_mapped_file = aFilePointer;

    return TRUE;
}
```

This function takes a string of opcodes and scans the target file for these bytes. If the opcodes cannot be found in the target, the location will be marked as changed. This is obviously a simple technique, but it works in many cases. Because of the problems listed at the beginning of this section, this approach can cause problems with false positives.

```
bool check_target_for_string(ea_t theAddress, DWORD theLen)
{
    bool ret = FALSE;
    if(theLen > 4096)
    {
        msg("skipping large buffer\n");
        return TRUE;
    }
```

```
try
{
    // Scan the target binary for the string.
    static char g_c[4096];

    // I don't know any other way to copy the data string
    // out of the IDA database?!
    for(DWORD i=0;i<theLen;i++)
    {
        g_c[i] = get_byte(theAddress + i);
    }
    // Here we have the opcode string; perform a search.
    LPVOID curr = g_mapped_file;
    DWORD sz = g_file_size;

    while(curr && sz)
    {
        LPVOID tp = memchr(curr, g_c[0], sz);
        if(tp)
        {
            sz -= ((char *)tp - (char *)curr);
        }

        if(tp && sz >= theLen)
        {
            if(0 == memcmp(tp, g_c, theLen))
            {
                // We found a match!
                ret = TRUE;
                break;
            }
            if(sz > 1)
            {
                curr = ((char *)tp)+1;
            }
            else
            {
                break;
            }
        }
        else
        {
            break;
        }
    }
```

```
    }
    catch(...)
    {
        msg("[!] critical failure.");
        return TRUE;
    }
    return ret;
}
```

This thread finds all the functions and compares them with a target binary:

```
void __cdecl _test(void *p)
{
    // Wait for start signal.
    while(g_tempest_state == 0)
    {
        Sleep(10);
    }
```

We call `get_func_qty()` to determine the number of functions in the loaded binary:

```
    ////////////////////////////////////
    // Enumerate through all functions.
    ////////////////////////////////////
    int total_functions = get_func_qty();
    int total_diff_matches = 0;
```

We now loop through each function. We call `getn_func()` to get the function structure for each function. The function structure is of type `func_t`. The `ea_t` type is known as "effective address" and is actually just an unsigned long. We get the start address of the function and the end address of the function from the function structure. We then compare the sequence of bytes with the target binary:

```
    for(int n=0;n<total_functions;n++)
    {
        // msg("getting next function \n");
        func_t *f = getn_func(n);
        ////////////////////////////////////////////
        // The start and end addresses of the function
        // are in the structure.
```

```
////////////////////////////////////////////
ea_t myea = f->startEA;
ea_t last_location = myea;

while((myea <= f->endEA) && (myea != BADADDR))
{
    // If the user has requested a stop we should return here.
    if(0 == g_tempest_state) return;

    ea_t nextea = get_first_cref_from(myea);
    ea_t amloc = get_first_cref_to(nextea);
    ea_t amloc2 = get_next_cref_to(nextea, amloc);

    // The cref will be the previous instruction, but we
    // also check for multiple references.
    if((amloc == myea) && (amloc2 == BADADDR))
    {
        // I was getting stuck in loops, so I added this hack
        // to force an exit to the next function.
        if(nextea > myea)
        {
            myea = nextea;

            // ---------------------------------------------
            // Uncomment the next two lines to get "cool"
            // scanning effect in the GUI. Looks sweet but slows
            // down the scan.
            // ---------------------------------------------
            // jumpto(myea);
            // refresh_idaview();
        }
        else myea = BADADDR;
    }
    else
    {
        // I am a location. Reference is not last instruction _OR_
        // I have multiple references.

        // Diff from the previous location to here and make a comment
        // if we don't match

        // msg("diffing location... \n");
```

We place a comment in our dead listing (using `add_long_cmt`) if the target doesn't contain our opcode string:

```
                bool pause_for_effect = FALSE;
                int size = myea - last_location;
                if(FALSE == check_target_for_string(last_location, size))
                {
                    add_long_cmt(last_location, TRUE,

                        "=====================================================\n" \
                        "= ** This code location differs from the
target  ** =\n" \

                        "=====================================================\n");
                    msg("Found location 0x%08X that didn't match
target!\n", last_location);
                    total_diff_matches++;
                }

                if(nextea > myea)
                {
                    myea = nextea;
                }
                else myea = BADADDR;

                // goto next address.
                jumpto(myea);
                refresh_idaview();
            }
        }
    }
    msg("Finished! Found %d locations that diff from the
    target.\n", total_diff_matches);
}
```

This function displays a dialog box prompting the user for a filename. This is a nice-looking dialog for file selection:

```
char * GetFilenameDialog(HWND theParentWnd)
{
    static TCHAR szFile[MAX_PATH] = "\0";

    strcpy( szFile, "");

    OPENFILENAME OpenFileName;
```

```
    OpenFileName.lStructSize = sizeof (OPENFILENAME);
    OpenFileName.hwndOwner = theParentWnd;
    OpenFileName.hInstance = GetModuleHandle("diff_scanner.plw");
    OpenFileName.lpstrFilter = "w00t! all files\0*.*\0\0";
    OpenFileName.lpstrCustomFilter = NULL;
    OpenFileName.nMaxCustFilter = 0;
    OpenFileName.nFilterIndex = 1;
    OpenFileName.lpstrFile = szFile;
    OpenFileName.nMaxFile = sizeof(szFile);
    OpenFileName.lpstrFileTitle = NULL;
    OpenFileName.nMaxFileTitle = 0;
    OpenFileName.lpstrInitialDir = NULL;
    OpenFileName.lpstrTitle = "Open";
    OpenFileName.nFileOffset = 0;
    OpenFileName.nFileExtension = 0;
    OpenFileName.lpstrDefExt = "*.*";
    OpenFileName.lCustData = 0;
    OpenFileName.lpfnHook           = NULL;
    OpenFileName.lpTemplateName  = NULL;
    OpenFileName.Flags = OFN_EXPLORER | OFN_NOCHANGEDIR;

    if(GetOpenFileName( &OpenFileName ))
    {
        return(szFile);
    }
    return NULL;
}
```

As with all "homegrown" dialogs, we need `DialogProc` to handle Windows messages:

```
BOOL CALLBACK MyDialogProc(HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch(msg)
    {
        case WM_COMMAND:
            if (LOWORD(wParam) == IDC_BROWSE)
            {
                char *p = GetFilenameDialog(hDlg);
                SetDlgItemText(hDlg, IDC_EDIT_FILENAME, p);
            }
            if (LOWORD(wParam) == IDC_START)
            {
                char filename[255];
                GetDlgItemText(hDlg, IDC_EDIT_FILENAME, filename, 254);
```

```
                if(0 == strlen(filename))
                {
                    MessageBox(hDlg, "You have not selected a target
file", "Try again", MB_OK);
                }
                else if(load_file(filename))
                {
                    g_tempest_state = 1;
                    EnableWindow( GetDlgItem(hDlg, IDC_START), FALSE);
                }
                else
                {
                    MessageBox(hDlg, "The target file could not be
opened", "Error", MB_OK);
                }
            }
            if (LOWORD(wParam) == IDC_STOP)
            {
                g_tempest_state = 0;
            }
            if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
            {
                if(LOWORD(wParam) == IDOK)
                {
                }
                EndDialog(hDlg, LOWORD(wParam));
                return TRUE;
            }
            break;
        default:
            break;
    }
    return FALSE;
}
void __cdecl _test2(void *p)
{
    DialogBox( GetModuleHandle("diff_scanner.plw"), MAKEINTRESOURCE(IDD_DIALOG1),
NULL, MyDialogProc);
}

//--------------------------------------------------------------------------
//
//   The plugin method.
//
//   This is the main function of plugin.
```

```
//
//   It will be called when the user selects the plugin.
//
//      Arg - the input argument. It can be specified in the
//         plugins.cfg file. The default is zero.
//
//
```

The `run` function is called when the user activates the plugin. In this case we start a couple threads and post a short message to the log window:

```
void run(int arg)
{
    // Testing.
    msg("starting diff scanner plugin\n");
    _beginthread(_test, 0, NULL);
    _beginthread(_test2, 0, NULL);
}
```

These global data items are used by IDA to display information about the plugin.

```
//--------------------------------------------------------------------------
char comment[] = "Diff Scanner Plugin, written by Greg Hoglund (www.rootkit.com)";
char help[] =
    "A plugin to find diffs in binary code\n"
    "\n"
    "This module highlights code locations that have changed.\n"
    "\n";

//--------------------------------------------------------------------------
// This is the preferred name of the plugin module in the menu system.
// The preferred name may be overridden in the plugins.cfg file.

char wanted_name[] = "Diff Scanner";

// This is the preferred hot key for the plugin module.
// The preferred hot key may be overridden in the plugins.cfg file.
// Note: IDA won't tell you if the hot key is not correct.
//    It will just disable the hot key.

char wanted_hotkey[] = "Alt-0";

//--------------------------------------------------------------------------
```

```
//
//   PLUGIN DESCRIPTION BLOCK
//
//----------------------------------------------------------------------

extern "C" plugin_t PLUGIN = {
 IDP_INTERFACE_VERSION,
 0,                // Plugin flags.
 init,             // Initialize.

 term,             // Terminate. this pointer may be NULL.

 run,              // Invoke plugin.

 comment,          // Long comment about the plugin
                   // It could appear in the status line
                   // or as a hint.

 help,             // Multiline help about the plugin

 wanted_name,      // The preferred short name of the plugin
 wanted_hotkey     // The preferred hot key to run the plugin
};
```

Decompiling and Disassembling Software

Decompilation is the process of transforming a binary executable—that is, a compiled program—into a higher level symbolic language that is easier for humans to understand. Usually this means turning a program executable into source code in a language like C. Most systems for decompiling can't directly convert programs into 100% source code. Instead, they usually provide an "almost there" kind of intermediate representation. Many reverse compilers are actually disassemblers that provide a dump of the machine code that makes a program work.

Probably the best decompiler available to the public is called IDA-Pro. IDA starts with a disassembly of program code and then analyzes program flow, variables, and function calls. IDA is hard to use and requires advanced knowledge of program behavior, but its technical level reflects the true nature of reverse engineering. IDA supplies a complete API for manipulating the program database so that users can perform custom analysis.

Other tools exist as well. A closed-source but free program called *REC* provides 100% C source code recovery for some kinds of binary executables. Another commercial disassembler is called *WDASM*. There are several decompilers for Java byte code that render Java source code (a process far less complicated than decompiling machine code for Intel chips). These systems tend to be very accurate, even when simple obfuscation techniques have been applied. There are open-source projects in this space as well, which interested readers can look up. It is always a good idea to keep several decompilers in your toolbox if you are interested in understanding programs.

Decompilers are used extensively in the computer underground to break copy protection schemes. This has given the tools an undeserved black eye. It is interesting to note that computer hacking and software piracy were largely independent in the early days of the computer underground. Hacking developed in UNIX environments, where software was free and source code was available, rendering decompiling somewhat unnecessary. Software piracy, on the other hand, was mainly developed to crack computer games, and hence was confined mainly to Apples, DOS, and Windows, for which source code was usually not available. The virus industry developed alongside the piracy movement. In the late 1990s, the hacking and cracking disciplines merged as more network software became available for Windows and hackers learned how to break Windows software. The current focus of decompiling is shifting from cracking copy protection to auditing software for exploitable bugs. The same old tricks are being used again, but in a new environment.

Decompilation in Practice: Reversing `helpctr.exe`

The following example illustrates a reverse engineering session against `helpctr.exe`, a Microsoft program provided with the Windows XP OS. The program happens to have a security vulnerability known as a *buffer overflow*. This particular vulnerability was made public quite some time ago, so revealing it here does not pose a real security threat. What is important for our purposes is describing the process of revealing the fault through reverse engineering. We use IDA-Pro to disassemble the target software. The target program produces a special debug file called a *Dr. Watson log*. We use only IDA and the information in the debug log to locate the exact coding error that caused the problem. Note that no source code is publicly available for the target software. Figure 3–4 shows IDA in action.

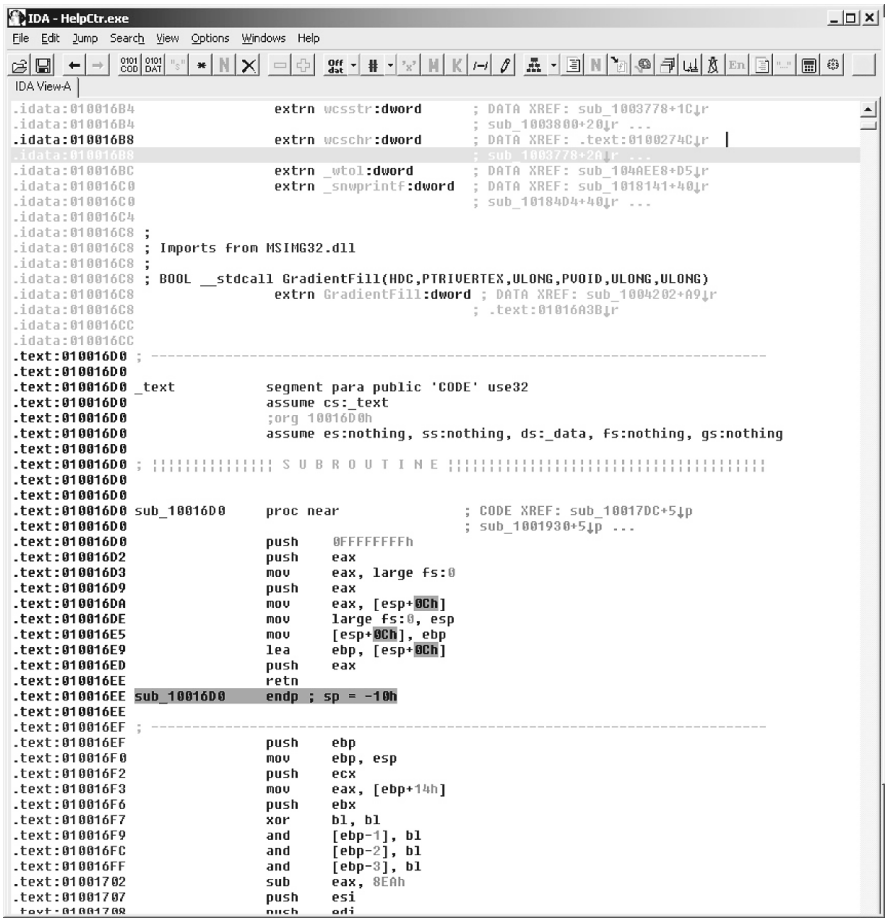


Figure 3–4 A screen shot of IDA-Pro reverse assembling the program `helpctr.exe`, which is included as part of the Microsoft Windows XP OS. As an exercise, we explore `helpctr.exe` for a buffer overflow vulnerability.

Bug Report

We learned of this vulnerability just like most people did, by reading a bug report posted to bugtraq, an industry mailing list forum where software problems and security issues are discussed. The report revealed only minor details about the problem. Most notably, the name of the executable and the input that caused the fault. The report revealed that the URL hcp://w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w.w., when supplied to Internet Explorer, caused `helpctr.exe` to launch. The URL does this by causing an application exception (which can be tickled remotely through a Web browser).

We recreate the fault by using the URL as input in a Windows XP environment. A debug log is created by the OS and we then copy the debug log and the `helpctr.exe` binary to a separate machine for analysis. Note that we used an older Windows NT machine to perform the analysis of this bug. The original XP environment is no longer required once we induce the error and gather the data we need.

The Debug Log

A debug dump is created when the program crashes. A stack trace is included in this log, giving us a hint regarding the location of the faulty code:

```
0006f8ac 0100b4ab 0006f8d8 00120000 00000103 msvcrt!wcsncat+0x1e
0006fae4 0050004f 00120000 00279b64 00279b44 HelpCtr+0xb4ab
0054004b 00000000 00000000 00000000 00000000 0x50004f
```

The culprit appears to be string concatenation function called `wcsncat`. The stack dump clearly shows our (fairly straightforward) URL string. We can see that the URL string dominates the stack space and thereby overflows other values:

```
*----> Raw Stack Dump <----*
000000000006f8a8 03 01 00 00 e4 fa 06 00 - ab b4 00 01 d8 f8 06 00 ................
000000000006f8b8 00 00 12 00 03 01 00 00 - d8 f8 06 00 a8 22 03 01 .............".."
000000000006f8c8 f9 00 00 00 b4 20 03 01 - cc 9b 27 00 c1 3e c4 77 ..... ....'..>.w
000000000006f8d8 43 00 3a 00 5c 00 57 00 - 49 00 4e 00 44 00 4f 00 C.:.\.W.I.N.D.O.
000000000006f8e8 57 00 53 00 5c 00 50 00 - 43 00 48 00 65 00 61 00 W.S.\.P.C.H.e.a.
000000000006f8f8 6c 00 74 00 68 00 5c 00 - 48 00 65 00 6c 00 70 00 l.t.h.\.H.e.l.p.
000000000006f908 43 00 74 00 72 00 5c 00 - 56 00 65 00 6e 00 64 00 C.t.r.\.V.e.n.d.
000000000006f918 6f 00 72 00 73 00 5c 00 - 77 00 2e 00 77 00 2e 00 o.r.s.\.w...w...
000000000006f928 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00 w...w...w...w...
000000000006f938 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00 w...w...w...w...
000000000006f948 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00 w...w...w...w...
000000000006f958 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00 w...w...w...w...
000000000006f968 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00 w...w...w...w...
000000000006f978 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00 w...w...w...w...
000000000006f988 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00 w...w...w...w...
000000000006f998 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00 w...w...w...w...
000000000006f9a8 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00 w...w...w...w...
000000000006f9b8 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00 w...w...w...w...
000000000006f9c8 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00 w...w...w...w...
000000000006f9d8 77 00 2e 00 77 00 2e 00 - 77 00 2e 00 77 00 2e 00 w...w...w...w...
```

Knowing that `wcsncat` is the likely culprit, we press onward with our analysis. Using IDA, we can see that `wcsncat` is called from two locations:

```
.idata:01001004         extrn wcsncat:dword   ; DATA XREF: sub_100B425+62□r
.idata:01001004                               ; sub_100B425+77□r ...
```

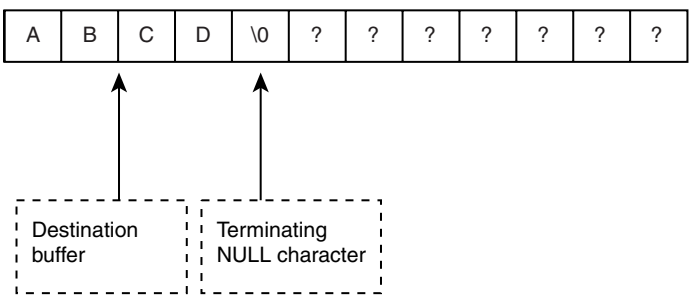
The behavior of `wcsncat` is straightforward and can be obtained from a manual. The call takes three parameters:

1. A destination buffer (a buffer pointer)
2. A source string (user supplied)
3. A maximum number of characters to append

The destination buffer is supposed to be large enough to store all the data being appended. (But note that in this case the data are supplied by an outside user, who might be malicious.) This is why the last argument lets the programmer specify the maximum length to append. Think of the buffer as a glass of a particular size, and the subroutine we're calling as a method for adding liquid to the glass. The last argument is supposed to guarantee that the glass does not overflow.

In `helpctr.exe`, a series of calls are made to `wcsncat` from within the broken subroutine. The following diagram illustrates the behavior of multiple calls to `wcsncat`. Assume the destination buffer is 12 characters long and we have already inserted the string `ABCD`. This leaves a total of eight remaining characters including the terminating NULL character.

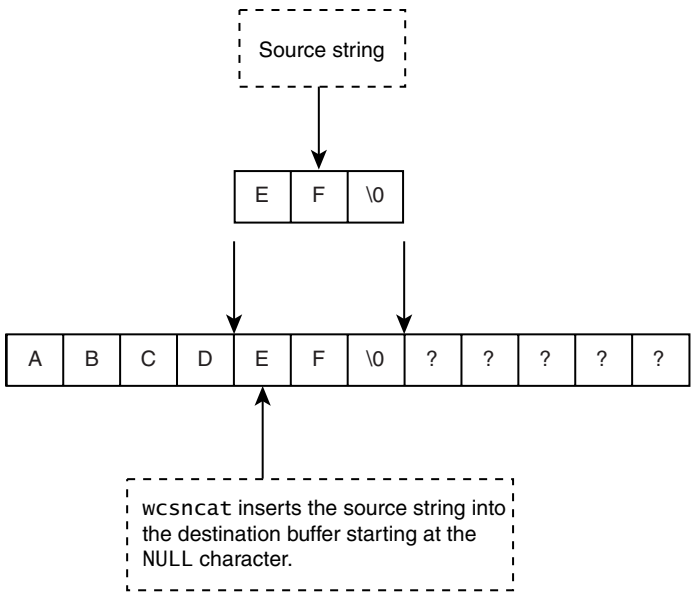
```
wcsncat(target_buffer, "ABCD", 11);
```



We now make a call to `wcsncat( )` and append the string `EF`. As the following diagram illustrates, the string is appended to the destination buffer starting at the NULL character. To protect the destination buffer, we must specify that a maximum of seven characters are to be appended. If the

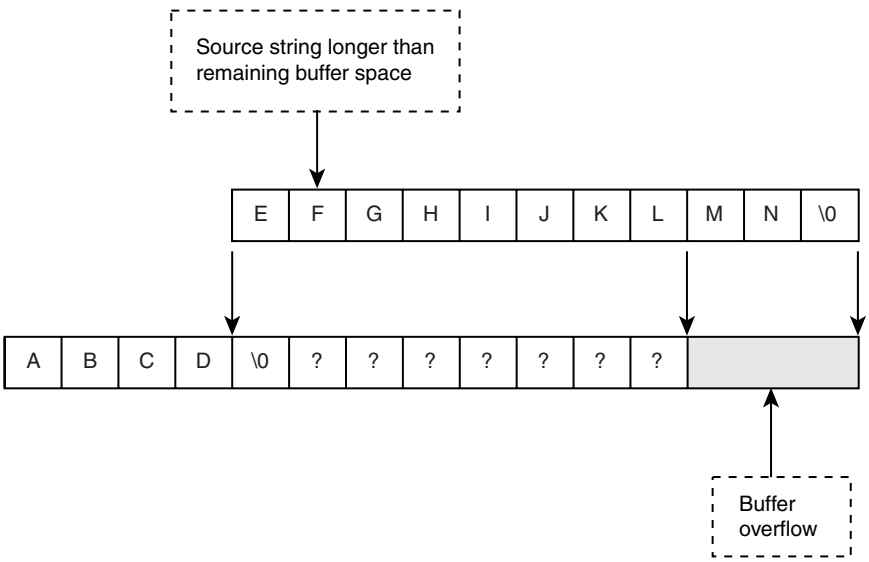
terminating NULL character is included, this makes a total of eight. Any more input will write off the end of our buffer and we will have a buffer overflow.

```
wcsncat(target_buffer, "EF", 7);
```



Unfortunately, in the faulty subroutine within `helpctr.exe`, the programmer made a subtle but fatal mistake. Multiple calls are made to `wscncat( )` but the maximum-length value is never recalculated. In other words, the multiple appends never account for the ever-shrinking space remaining at the end of the destination buffer. The glass is getting full, but nobody is watching as more liquid is poured in. In our illustration, this would be something like appending `EFGHIJKLMN` to our example buffer, using the maximum length of 11 characters (12 including the `NULL`). The correct value should be a maximum of seven characters, but we never correct for this and we append past the end of our buffer.

```
wcsncat(target_buffer, "EFGHIJKLMN", 11);
```



A graph of the subroutine in `helpctr.exe` that makes these calls is shown in Figure 3–5.

A very good reverse engineer can spot and decode the logic that causes this problem in 10 to 15 minutes. An average reverse engineer might be able to reverse the routine in about an hour. The subroutine starts out by checking that it has not been passed a `NULL` buffer. This is the first JZ branch. If the buffer is valid, we can see that `103h` is being set in a register. This is 259 decimal—meaning we have a maximum buffer size of 259 characters.[13] And herein lies the bug. We see that this value is never updated during successive calls to `wcsncat`. Strings of characters are appended to the target buffer multiple times, but the maximum allowable length is never appropriately reduced. This type of bug is very typical of parsing problems often found in code. Parsing typically includes lexical and syntax analysis of user-supplied strings, but it unfortunately often fails to maintain proper buffer arithmetic.

What is the final conclusion here? A user-supplied variable—in the URL used to spawn `helpctr.exe`—is passed down to this subroutine, which subsequently uses the data in a buggy series of calls for string concatenation.

13. The actual buffer size is double (518 bytes), because we are working with wide characters. This is not important to the current discussion, however.

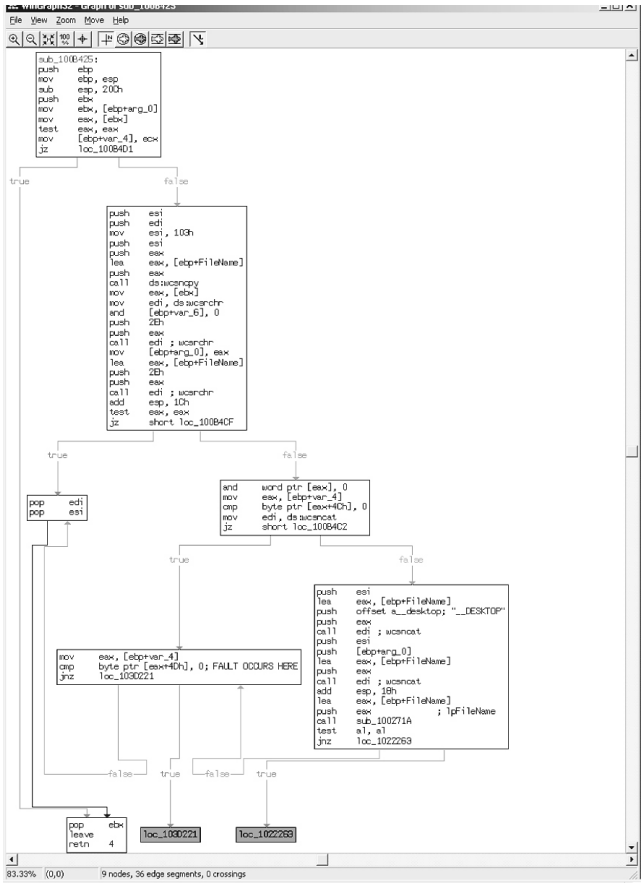


Figure 3–5 A simple graph of the subroutine in `helpctr.exe` that makes calls to `wcsncat()`.

Alas, yet another security problem in the world caused by sloppy code. We leave an exploit resulting in machine compromise as an exercise for you to undertake.

Automatic, Bulk Auditing for Vulnerabilities

Clearly, reverse engineering is a time-consuming task and a process that does not scale well. There are many cases when reverse engineering for security bugs would be valuable, but there isn't nearly enough time to analyze each and every component of a software system the way we have done in the previous section. One possibility, however, is automated analysis. IDA

provides a platform for adding your own analysis algorithms. By writing a special script for IDA, we can automate some of the tasks required for finding a vulnerability. Here, we provide an example of strict white box analysis.[14]

Harking back to a previous example, let's assume we want to find other bugs that may involve the (mis)use of `wcsncat`. We can use a utility called `dumpbin` under Windows to show which calls are imported by an executable:

```
dumpbin /imports target.exe
```

To bulk audit all the executables on a system, we can write a small Perl script. First create a list of executables to analyze. Use the `dir` command as follows:

```
dir /B /S c:\winnt\*.exe > files.txt
```

This creates a large output file of all the executable files under the WINNT directory. The Perl script will then call `dumpbin` on each file and will analyze the results to determine whether `wscncat` is being used:

```
open(FILENAMES, "files.txt");
while (<FILENAMES>)
{
    chop($_);
    my $filename = $_;
    $command = "dumpbin /imports $_ > dumpfile.txt";
    #print "trying $command";
    system($command);

    open(DUMPFILE, "dumpfile.txt");
    while (<DUMPFILE>)
    {
        if(m/wcsncat/gi)
        {
            print "$filename: $_";
        }
    }
```

14. The reason this is a white box analysis (and not a black box analysis) is that we're looking "inside" the program to find out what's happening. Black box approaches treat a target program as an opaque box that can only be probed externally. White box approaches dive into the box (regardless of whether source code is available).

```
    close(DUMPFILE);
}
close(FILENAMES);
```

Running this script on a system in the lab produces the following output:

```
C:\temp>perl scan.pl
c:\winnt\winrep.exe:    7802833F  2E4 wcsncat
c:\winnt\INF\UNREGMP2.EXE:    78028EDD  2E4 wcsncat
c:\winnt\SPEECH\VCMD.EXE:    78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\dfrgfat.exe:    77F8F2A0  499 wcsncat
c:\winnt\SYSTEM32\dfrgntfs.exe:    77F8F2A0  499 wcsncat
c:\winnt\SYSTEM32\IESHWIZ.EXE:    78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\NET1.EXE:    77F8E8A2  491 wcsncat
c:\winnt\SYSTEM32\NTBACKUP.EXE:    77F8F2A0  499 wcsncat
c:\winnt\SYSTEM32\WINLOGON.EXE:          2E4 wcsncat
```

We can see that several of the programs under Windows NT are using `wcsncat`. With a little time we can audit these files to determine whether they suffer from similar problems to the example program we show earlier. We could also examine DLLs using this method and generate a much larger list:

```
C:\temp>dir /B /S c:\winnt\*.dll > files.txt

C:\temp>perl scan.pl

c:\winnt\SYSTEM32\AAAAMON.DLL:    78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\adsldpc.dll:    7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\avtapi.dll:    7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\AVWAV.DLL:    78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\BR549.DLL:    78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\CMPROPS.DLL:    78028EDD  2E7 wcsncat
c:\winnt\SYSTEM32\DFRGUI.DLL:    78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\dhcpmon.dll:    7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\dmloader.dll:          2FB wcsncat
c:\winnt\SYSTEM32\EVENTLOG.DLL:    78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\GDI32.DLL:    77F8F2A0  499 wcsncat
c:\winnt\SYSTEM32\IASSAM.DLL:    78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\IFMON.DLL:    78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\LOCALSPL.DLL:    7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\LSASRV.DLL:          2E4 wcsncat
c:\winnt\SYSTEM32\mpr.dll:    77F8F2A0  499 wcsncat
c:\winnt\SYSTEM32\MSGINA.DLL:    7802833F  2E4 wcsncat
```

```
c:\winnt\SYSTEM32\msjetoledb40.dll:    7802833F  2E2 wcsncat
c:\winnt\SYSTEM32\MYCOMPUT.DLL:    78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\netcfgx.dll:    7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\ntdsa.dll:    7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\ntdsapi.dll:    7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\ntdsetup.dll:    7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\ntmssvc.dll:    7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\NWWKS.DLL:    7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\ODBC32.dll:    7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\odbccp32.dll:    7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\odbcjt32.dll:    7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\OIPRT400.DLL:    78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\PRINTUI.DLL:    7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\rastls.dll:    7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\rend.dll:    7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\RESUTILS.DLL:    7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\SAMSRV.DLL:    7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\scecli.dll:    7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\scesrv.dll:    7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\sqlsrv32.dll:         2E2 wcsncat
c:\winnt\SYSTEM32\STI_CI.DLL:    78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\USER32.DLL:    77F8F2A0  499 wcsncat
c:\winnt\SYSTEM32\WIN32SPL.DLL:    7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\WINSMON.DLL:    78028EDD  2E4 wcsncat
c:\winnt\SYSTEM32\dllcache\dmloader.dll:         2FB wcsncat
c:\winnt\SYSTEM32\SETUP\msmqocm.dll:    7802833F  2E4 wcsncat
c:\winnt\SYSTEM32\WBEM\cimwin32.dll:    7802833F  2E7 wcsncat
c:\winnt\SYSTEM32\WBEM\WBEMCNTL.DLL:    78028EDD  2E7 wcsncat
```

Batch Analysis with IDA-Pro

We already illustrated how to write a plugin module for IDA. IDA also supports a scripting language. The scripts are called *IDC scripts* and can sometimes be easier than using a plugin. We can perform a batch analysis with the IDA-Pro tool by using an IDC script as follows:

```
c:\ida\idaw -Sbatch_hunt.idc -A -c c:\winnt\notepad.exe
```

with the very basic IDC script file shown here:

```
#include <idc.idc>
//--------------------------------------------------------------------
static main(void) {
 Batch(1);
```

```
 /* will hang if existing database file */
 Wait();
 Exit(0);
}
```

As another example, consider batch analysis for `sprintf` calls. The Perl script calls IDA using the command line:

```
open(FILENAMES, "files.txt");
while (<FILENAMES>)
{
    chop($_);
    my $filename = $_;
    $command = "dumpbin /imports $_ > dumpfile.txt";
    #print "trying $command";

    system($command);

    open(DUMPFILE, "dumpfile.txt");
    while (<DUMPFILE>)
    {
        if(m/sprintf/gi)
        {
            print "$filename: $_\n";
            system("c:\\ida\\idaw -Sbulk_audit_sprintf.idc -A -c $filename");
        }
    }
    close(DUMPFILE);
}
close(FILENAMES);
```

We use the script `bulk_audit_sprintf.idc`:

```
//
//   This example shows how to use GetOperandValue() function.
//

#include <idc.idc>

/* this routine is hard coded to understand sprintf calls */

static hunt_address(     eb,         /* the address of this call */
            param_count,  /* the number of parameters for this call */
            ec,         /* maximum number of instructions to backtrace */
```

```
            output_file
            )
{
    auto ep; /* placeholder */
    auto k;
    auto kill_frame_sz;
    auto comment_string;

    k = GetMnem(eb);

    if(strstr(k, "call") != 0)
    {
        Message("Invalid starting point\n");
        return;
    }

    /* backtrace code */
    while( eb=FindCode(eb, 0) )
    {
        auto j;
        j = GetMnem(eb);

        /* exit early if we run into a retn code */
        if(strstr(j, "retn") == 0) return;

        /* push means argument to sprintf call */
        if(strstr(j, "push") == 0)
        {
            auto my_reg;
            auto max_backtrace;

            ep = eb; /* save our place */

            /* work back to find out the parameter */
            my_reg = GetOpnd(eb, 0);
            fprintf(output_file, "push number %d, %s\n", param_count, my_reg);

            max_backtrace = 10; /* don't backtrace more than 10 steps */
            while(1)
            {
                auto x;
                auto y;

                eb = FindCode(eb, 0); /* backwards */
```

```
                x = GetOpnd(eb,0);
                if ( x != -1 )
                {
                    if(strstr(x, my_reg) == 0)
                    {
                        auto my_src;
                        my_src = GetOpnd(eb, 1);

                        /* param 3 is the target buffer */
                        if(3 == param_count)
                        {
                            auto my_loc;
                            auto my_sz;
                            auto frame_sz;

                            my_loc = PrevFunction(eb);

                            fprintf(output_file, "detected
subroutine 0x%x\n", my_loc);

                            my_sz = GetFrame(my_loc);
                            fprintf(output_file, "got frame
%x\n", my_sz);

                            frame_sz = GetFrameSize(my_loc);
                            fprintf(output_file, "got frame size
%d\n", frame_sz);

                            kill_frame_sz =
GetFrameLvarSize(my_loc);
                            fprintf(output_file, "got frame lvar
size %d\n", kill_frame_sz);

                            my_sz = GetFrameArgsSize(my_loc);
                            fprintf(output_file, "got frame args
size %d\n", my_sz);

                            /* this is the target buffer */
                            fprintf(output_file, "%s is the
target buffer, in frame size %d bytes\n", my_src, frame_sz);
                        }

                        /* param 1 is the source buffer */
                        if(1 == param_count)
                        {
```

```
                            fprintf(output_file, "%s is the
source buffer\n", my_src);
                            if(-1 != strstr(my_src, "arg"))
                            {
                                fprintf(output_file, "%s is
an argument that will overflow if larger than %d bytes!\n", my_src, kill_frame_sz);
                            }
                        }
                        break;
                    }
                }
                max_backtrace--;
                if(max_backtrace == 0)break;
            }
            eb = ep; /* reset to where we started and continue for next
parameter */
            param_count--;
            if(0 == param_count)
            {
                fprintf(output_file, "Exhausted all  parameters\n");
                return;
            }
        }
        if(ec-- == 0)break; /* max backtrace looking for parameters */
    }
}

static main()
{
    auto ea;
    auto eb;
    auto last_address;
    auto output_file;
    auto file_name;

    /* turn off all dialog boxes for batch processing */
    Batch(0);
     /* wait for autoanalysis to complete */
     Wait();

    ea = MinEA();
    eb = MaxEA();

    output_file = fopen("report_out.txt", "a");
```

```
    file_name = GetIdbPath();

    fprintf(output_file, "-----------------------------------------------\nFilename:
%s\n", file_name);
    fprintf(output_file, "HUNTING FROM %x TO %x\n-----------------------------------
----------\n", ea, eb);
    while(ea != BADADDR)
    {
        auto my_code;

        last_address=ea;
        //Message("checking %x\n", ea);
        my_code = GetMnem(ea);
        if(0 == strstr(my_code, "call")){
            auto my_op;
            my_op = GetOpnd(ea, 0);
            if(-1 != strstr(my_op, "sprintf")){
                fprintf(output_file, "Found sprintf call at 0x%x -
checking\n", ea);

                /* 3 parameters, max backtrace of 20 */
                hunt_address(ea, 3, 20, output_file);
                fprintf(output_file, "-----------------------------------
----------\n");
            }
        }
        ea = FindCode(ea, 1);
    }
    fprintf(output_file, "FINISHED at address 0x%x\n---------------------------------
-------------\n", last_address);
    fclose(output_file);
    Exit(0);
}
```

The output produced by this simple batch file is placed in a file called report_out.txt for later analysis. The file looks something like this:

```
-----------------------------------------------
Filename: C:\reversing\of1.idb
HUNTING FROM 401000 TO 404000
-----------------------------------------------
Found sprintf call at 0x401012 - checking
push number 3, ecx
detected subroutine 0x401000
```

```
got frame ff00004f
got frame size 32
got frame lvar size 28
got frame args size 0
[esp+1Ch+var_1C] is the target buffer, in frame size 32 bytes
push number 2, offset unk_403010
push number 1, eax
[esp+arg_0] is the source buffer
[esp+arg_0] is an argument that will overflow if larger than 28 bytes!
Exhausted all parameters
-----------------------------------------------
Found sprintf call at 0x401035 - checking
push number 3, ecx
detected subroutine 0x401020
got frame ff000052
got frame size 292
got frame lvar size 288
got frame args size 0
[esp+120h+var_120] is the target buffer, in frame size 292 bytes
push number 2, offset aSHh
push number 1, eax
[esp+arg_0] is the source buffer
[esp+arg_0] is an argument that will overflow if larger than 288 bytes!
Exhausted all parameters
-----------------------------------------------
FINISHED at address 0x4011b6
-----------------------------------------------
-----------------------------------------------
Filename: C:\winnt\MSAGENT\AGENTCTL.idb
HUNTING FROM 74c61000 TO 74c7a460
-----------------------------------------------
Found sprintf call at 0x74c6e3b6 - checking
push number 3, eax
detected subroutine 0x74c6e2f9
got frame ff000eca
got frame size 568
got frame lvar size 552
got frame args size 8
[ebp+var_218] is the target buffer, in frame size 568 bytes
push number 2, offset aD__2d
push number 1, eax
[ebp+var_21C] is the source buffer
Exhausted all parameters
-----------------------------------------------
```

Searching the function calls, we see a suspect call to `lstrcpy()`. Analyzing lots of code automatically is a common trick to look for good starting places, and it turns out to be very useful in practice.

Writing Your Own Cracking Tools

Reverse engineering is mostly a tedious sport consisting of thousands of small steps and encompassing bazillions of facts. The human mind cannot manage all the data needed to do this in a reasonable way. If you're like most people, you are going to need tools to help you manage all the data. There are quite a number of debugging tools available on the market and in freeware form, but sadly most of them do not present a complete solution. For this reason, you are likely to need to write your own tools.

Coincidentally, writing tools is a great way to learn about software. Writing tools requires a real understanding of the architecture of software—most important, how software tends to be structured in memory and how the heap and stack operate. Learning by writing tools is more efficient than a blind brute-force approach using pencil and paper. Your skills will be better honed by tool creation, and the larval stage (learning period) will not take as long.

x86 Tools

The most common processor in most workstations seems to be the Intel x86 family, which includes the 386, 486, and Pentium chips. Other manufacturers also make compatible chips. The chips are a family because they have a subset of features that are common to all the processors. This subset is called the *x86 feature set*. A program that is running on an x86 processor will usually have a stack, a heap, and a set of instructions. The x86 processor has registers that contain memory addresses. These addresses indicate the location in memory where important data structures reside.

The Basic x86 Debugger

Microsoft supplies a relatively easy-to-use debugging API for Windows. The API allows you to access debugging events from a user-mode program using a simple loop. The structure of the program is quite simple:

```
DEBUG_EVENT    dbg_evt;
    m_hProcess = OpenProcess(    PROCESS_ALL_ACCESS | PROCESS_VM_OPERATION,
                                        0,
```

```
                                        mPID);
    if(m_hProcess == NULL)
    {
        _error_out("[!] OpenProcess Failed !\n");
        return;
    }

    // Alright, we have the process opened; time to start debugging.
    if(!DebugActiveProcess(mPID))
    {
        _error_out("[!] DebugActiveProcess failed !\n");
        return;
    }

    // Don't kill the process on thread exit.
    // Note: only supported on Windows XP.
    fDebugSetProcessKillOnExit(FALSE);

    while(1)
    {
        if(WaitForDebugEvent(&dbg_evt, DEBUGLOOP_WAIT_TIME))
        {
            // Handle the debug events.
            OnDebugEvent(dbg_evt);

            if(!ContinueDebugEvent(    mPID,
dbg_evt.dwThreadId,
DBG_CONTINUE))
            {
                _error_out("ContinueDebugEvent failed\n");
                break;
            }
        }
        else
        {
            // Ignore timeout errors.
            int err = GetLastError();
            if(121 != err)
            {
                _error_out("WaitForDebugEvent failed\n");
                break;
            }
        }

        // Exit if debugger has been disabled.
```

```
        if(FALSE == mDebugActive)
        {
            break;
        }
    }

    RemoveAllBreakPoints();
```

This code shows how you can connect to an already running process. You can also launch a process in debug mode. Either way, the debugging loop is the same: You simply wait for debug events. The loop continues until there is an error or the `mDebugActive` flag is set to `TRUE`. In either case, once the debugger exits, the debugger is automatically detached from the process. If you are running on Windows XP, the debugger is detached gracefully and the target process can continue executing. If you are on an older version of Windows, the debugger API will kill the patient (the target process dies). In fact, it is considered quite annoying that the debugger API kills the target process on detach! In some people's opinion this was a serious design flaw of the Microsoft debugging API that should have been fixed in version 0.01. Fortunately, this has finally been fixed in the Windows XP version.

On Breakpoints

Breakpoints are central to debugging. Elsewhere in the book you will find references to standard breakpoint techniques. A breakpoint can be issued using a simple instruction. The standard breakpoint instruction under x86 seems to be interrupt 3. The nice thing about interrupt 3 is that it can be coded as a single byte of data. This means it can be patched over existing code with minimal concern for the surrounding code bytes. This breakpoint is easy to set in code by copying the original byte to a safe location and replacing it with the byte `0xCC`.

Breakpoint instructions are sometimes globbed together into blocks and are written to invalid regions of memory. Thus, if the program "accidentally" jumps to one of these invalid locations, the debug interrupt will fire. You sometimes see this on the program stack in regions between stack frames.

Of course, interrupt 3 doesn't *have* to be the way a breakpoint is handled. It could just as easily be interrupt 1, or anything for that matter. The interrupts are software driven and the software of the OS decides how it will handle the event. This is controlled via the interrupt descriptor table

(when the processor is running in protected mode) or the interrupt vector table (when running in real mode).

To set a breakpoint, you must first save the original instruction you are replacing, then when you remove the breakpoint you can put the saved instruction back in its original location. The following code illustrates saving the original value before setting a breakpoint:

```
//////////////////////////////////////////////////////////////////////////////
// Change the page protection so we can read the original target instruction,
// then change it back when we are done.
//////////////////////////////////////////////////////////////////////////////
MEMORY_BASIC_INFORMATION mbi;
VirtualQueryEx( m_hProcess,
            (void *)(m_bp_address),
            &mbi,
            sizeof(MEMORY_BASIC_INFORMATION));

// Now read the original byte.
if(!ReadProcessMemory(    m_hProcess,
                  (void *)(m_bp_address),
                  &(m_original_byte),
                  1,
                  NULL))
{
    _error_out("[!] Failed to read process memory ! \n");
    return NULL;
}

if(m_original_byte == 0xCC)
{
    _error_out("[!] Multiple setting of the same breakpoint ! \n");
    return NULL;
}

DWORD dwOldProtect;
// Change protection back.
if(!VirtualProtectEx( m_hProcess,
                  mbi.BaseAddress,
                  mbi.RegionSize,
                  mbi.Protect,
                  &dwOldProtect ))
{
    _error_out("VirtualProtect failed!");
    return NULL;
```

```
    }

    SetBreakpoint();
```

The previous code alters the memory protection so we can read the target address. It stores the original data byte. The following code then overwrites the memory with a `0xCC` instruction. Notice that we check the memory to determine whether a breakpoint was already set before we arrived.

```
bool SetBreakpoint()
{
    char a_bpx = '\xCC';

    if(!m_hProcess)
    {
        _error_out("Attempt to set breakpoint without target process");
        return FALSE;
    }

    ////////////////////////////////////////////////////////////////////////////
    // Change the page protection so we can write, then change it back.
    ////////////////////////////////////////////////////////////////////////////
    MEMORY_BASIC_INFORMATION mbi;
    VirtualQueryEx( m_hProcess,
                    (void *)(m_bp_address),
                    &mbi,
                    sizeof(MEMORY_BASIC_INFORMATION));

    if(!WriteProcessMemory(m_hProcess, (void *)(m_bp_address), &a_bpx, 1, NULL))
    {
        char _c[255];
        sprintf(_c,
            "[!] Failed to write process memory, error %d ! \n", GetLastError());
        _error_out(_c);
        return FALSE;
    }

    if(!m_persistent)
    {
        m_refcount++;
    }

    DWORD dwOldProtect;
    // Change protection back.
```

```
    if(!VirtualProtectEx(     m_hProcess,
                              mbi.BaseAddress,
                              mbi.RegionSize,
                              mbi.Protect,
                              &dwOldProtect ))
    {
        _error_out("VirtualProtect failed!");
        return FALSE;
    }

    // TODO: Flush instruction cache.

    return TRUE;
}
```

The previous code writes to the target process memory a single `0xCC` byte. As an instruction, this is translated as an interrupt 3. We must first change the page protection of the target memory so that we can write to it. We change the protection back to the original value before allowing the program to continue. The API calls used here are fully documented in Microsoft Developer Network (MSDN) and we encourage you to check them out there.

Reading and Writing Memory

Once you have hit a breakpoint, the next task is usually to examine memory. If you want to use some of the debugging techniques discussed in this book you need to examine memory for user-supplied data. Reading and writing to memory is easily accomplished in the Windows environment using a simple API. You can query to see what kind of memory is available and you can also read and write memory using routines that are similar to `memcpy`.

If you want to query a memory location to determine whether it's valid or what properties are set (read, write, nonpaged, and so on) you can use the `VirtualQueryEx` routine.

```
//////////////////////////////////////////////////////
// Check that we can read the target memory address.
//////////////////////////////////////////////////////
bool can_read( CDThread *theThread, void *p )
{
    bool ret = FALSE;
```

```
    MEMORY_BASIC_INFORMATION mbi;

    int sz =
    VirtualQueryEx( theThread->m_hProcess,
                                (void *)p,
                                &mbi,
                                sizeof(MEMORY_BASIC_INFORMATION));

    if(    (mbi.State == MEM_COMMIT)
            &&
            (mbi.Protect != PAGE_READONLY)
            &&
            (mbi.Protect != PAGE_EXECUTE_READ)
            &&
            (mbi.Protect != PAGE_GUARD)
            &&
            (mbi.Protect != PAGE_NOACCESS)
            )
    {
        ret = TRUE;
    }
    return ret;
}
```

The example function will determine whether the memory address is readable. If you want to read or write to memory you can use the `ReadProcessMemory` and `WriteProcessMemory` API calls.

Debugging Multithreaded Programs

If the program has multiple threads, you can control the behavior of each individual thread (something that is very helpful when attacking more modern code). There are API calls for manipulating the thread. Each thread has a `CONTEXT`. A context is a data structure that controls important process data like the current instruction pointer. By modifying and querying context structures, you can control and track all the threads of a multithreaded program. Here is an example of setting the instruction pointer of a given thread:

```
bool SetEIP(DWORD theEIP)
{
    CONTEXT ctx;
    HANDLE hThread =
    fOpenThread(
```

```
            THREAD_ALL_ACCESS,
            FALSE,
            m_thread_id
            );

    if(hThread == NULL)
    {
        _error_out("[!] OpenThread failed ! \n");
        return FALSE;
    }

    ctx.ContextFlags = CONTEXT_FULL;
    if(!::GetThreadContext(hThread, &ctx))
    {
        _error_out("[!] GetThreadContext failed ! \n");
        return FALSE;
    }

    ctx.Eip = theEIP;
    ctx.ContextFlags = CONTEXT_FULL;
    if(!::SetThreadContext(hThread, &ctx))
    {
        _error_out("[!] SetThreadContext failed ! \n");
        return FALSE;
    }

    CloseHandle(hThread);

    return TRUE;
}
```

From this example you can see how to read and set the thread context structure. The thread context structure is fully documented in the Microsoft header files. Note that the context flag `CONTEXT_FULL` is set during a `get` or `set` operation. This allows you to control all the data values of the thread context structure.

Remember to close your thread handle when you are finished with the operation or else you will cause a resource leak problem. The example uses an API call called `OpenThread`. If you cannot link your program to `OpenThread` you will need to import the call manually. This has been done in the example, which uses a function pointer named `fOpenThread`. To initialize `fOpenThread` you must import the function pointer directly from `KERNEL32.DLL`:

```
typedef
void *
(__stdcall *FOPENTHREAD)
(
 DWORD dwDesiredAccess, // Access right
 BOOL bInheritHandle,   // Handle inheritance option
 DWORD dwThreadId       // Thread identifier
);

FOPENTHREAD fOpenThread=NULL;

fOpenThread = (FOPENTHREAD)
    GetProcAddress(
            GetModuleHandle("kernel32.dll"),
            "OpenThread" );
            if(!fOpenThread)
            {
                _error_out("[!] failed to get openthread function!\n");
             }
```

This is a particularly useful block of code because it illustrates how to define a function and import it from a DLL manually. You may use variations of this syntax for almost any exported DLL function.

Enumerate Threads or Processes

Using the "toolhelp" API that is supplied with Windows you can query all running processes and threads. You can use this code to query all running threads in your debug target.

```
    // For the target process, build a
    // thread structure for each thread.

    HANDLE         hProcessSnap = NULL;
    hProcessSnap = CreateToolhelp32Snapshot(
                   TH32CS_SNAPTHREAD,
                   mPID);
    if (hProcessSnap == INVALID_HANDLE_VALUE)
    {
        _error_out("toolhelp snap failed\n");
      return;
    }
    else
    {
        THREADENTRY32 the;
```

```
        the.dwSize = sizeof(THREADENTRY32);

        BOOL bret = Thread32First( hProcessSnap, &the);
        while(bret)
        {
            // Create a thread structure.
            if(the.th32OwnerProcessID == mPID)
            {
                CDThread *aThread = new CDThread;
                aThread->m_thread_id = the.th32ThreadID;
                aThread->m_hProcess = m_hProcess;

                mThreadList.push_back( aThread );
            }
            bret = Thread32Next(hProcessSnap, &the);
        }
    }
```

In this example, a `CDThread` object is being built and initialized for each thread. The thread structure that is obtained, THREADENTRY32, has many interesting values to the debugger. We encourage you to reference the Microsoft documentation on this API. Note that the code checks the owner process identification (PID) for each thread to make sure it belongs to the debug target process.

Single Stepping

Tracing the flow of program execution is very important when you want to know if the attacker (or maybe you) can control logic. For example, if the 13th byte of the packet is being passed to a switch statement, the attacker controls the switch statement by virtue of the fact that the attacker controls the 13th byte of the packet.

Single stepping is a feature of the x86 chipset. There is a special flag (called `TRAP FLAG`) in the processor that, if set, will cause only a single instruction to be executed followed by an interrupt. Using the single-step interrupt, a debugger can examine each and every instruction that is executing. You can also examine memory at each step using the routines listed earlier. In fact, this is exactly what a tool called *The PIT* does.[15] These techniques are all fairly simple, but when properly combined, they result in a very powerful debugger.

15. The PIT tool is available at http://www.hbgary.com.

To put the processor into single step, you must set the single-step flag. The following code illustrates how to do this:

```
bool SetSingleStep()
{
    CONTEXT ctx;

    HANDLE hThread =
        fOpenThread(
                THREAD_ALL_ACCESS,
                FALSE,
                m_thread_id
                );

    if(hThread == NULL)
    {
        _error_out("[!] Failed to Open the BPX thread !\n");
        return FALSE;
    }

    // Rewind one instruction. This means no manual snapshots anymore.
    ctx.ContextFlags = CONTEXT_FULL;
    if(!::GetThreadContext(hThread, &ctx))
    {
        _error_out("[!] GetThreadContext failed ! \n");
        return FALSE;
    }
    // Set single step for this thread.
    ctx.EFlags |= TF_BIT ;
    ctx.ContextFlags = CONTEXT_FULL;
    if(!::SetThreadContext(hThread, &ctx))
    {
        _error_out("[!] SetThreadContext failed ! \n");
        return FALSE;
    }

    CloseHandle(hThread);
    return TRUE;
}
```

Note that we influence the `trace` flag by using the thread context structures. The thread ID is stored in a variable called `m_thread_id`. To single step a multithreaded program, all threads must be set single step.

Patching

If you are using our kind of breakpoints, you have already experienced patching. By reading the original byte of an instruction and replacing it with `0xCC`, you patched the original program! Of course the technique can be used to patch in much more than a single instruction. Patching can be used to insert branching statements, new code blocks, and even to overwrite static data. Patching is one way that software pirates have cracked digital copyright mechanisms. In fact, many interesting things are made possible by changing only a single jump statement. For example, if a program has a block of code that checks the license file, all the software pirate needs to do is insert a jump that branches around the license check.[16] If you are interested in software cracking, there are literally thousands of documents on the Net published on the subject. These are easily located on the Internet by googling "software cracking."

Patching is an important skill to learn. It allows you, in many cases, to fix a software bug. Of course, it also allows you to *insert* a software bug. You may know that a certain file is being used by the server software of your target. You can insert a helpful backdoor using patching techniques. There is a good example of a software patch (patching the NT kernel) discussed in Chapter 8.

Fault Injection

Fault injection can take many forms [Voas and McGraw, 1999]. At its most basic, the idea is simply to supply strange or unexpected inputs to a software program and see what happens. Variations of the technique involve mutating the code and injecting corruption into the data heap or program stack. The goal is to cause the software to fail in interesting ways.

Using fault injection, software will *always* fail. The question is *how* does it fail? Does the software fail in a way that allows an attacker to gain access to the system? Does the software reveal secret information? Does the failure result in a cascade failure that affects other parts of the system? Failures that do not cause damage to the system indicate a fault-tolerant system.

Fault injection is one of the most powerful testing methodologies ever invented, yet it remains one of the most underused by commercial software vendors. This is one of the reasons why commercial software has so many

16. This very basic approach is no longer used much in practice. More complicated schemes are discussed in *Building Secure Software* [Viega and McGraw, 2001].

bugs today. Many so-called software engineers subscribe to the philosophy that a rigid software development process necessarily results in secure and bug-free code, but it ain't necessarily so. The real world has shown us repeatedly that without a solid testing strategy, code will always have dangerous bugs. It's almost amusing (from an attacker's perspective) to know that software testing is still receiving the most meager of budgets in most software houses today. This means the world will belong to the attackers for many years to come.

Fault injection on software input is a good way to test for vulnerabilities. The reason is simple: The attacker controls the software input, so it's natural to test every possible input combination that an attacker can supply. Eventually you are bound to find a combination that exploits the software, right?![17]

Process Snapshots

When a breakpoint fires, the program becomes frozen in mid run. All execution in all threads is stopped. It is possible at this point to use the memory routines to read or write any part of the program memory. A typical program will have several relevant memory sections. This is a snapshot of memory from the named server running BIND 9.02 under Windows NT:

```
named.exe:
Found memory based at 0x00010000, size 4096
Found memory based at 0x00020000, size 4096
Found memory based at 0x0012d000, size 4096
Found memory based at 0x0012e000, size 8192
Found memory based at 0x00140000, size 184320
Found memory based at 0x00240000, size 24576
Found memory based at 0x00250000, size 4096
Found memory based at 0x00321000, size 581632
Found memory based at 0x003b6000, size 4096
Found memory based at 0x003b7000, size 4096
Found memory based at 0x003b8000, size 4096
Found memory based at 0x003b9000, size 12288
Found memory based at 0x003bc000, size 8192
Found memory based at 0x003be000, size 8192
Found memory based at 0x003c0000, size 8192
Found memory based at 0x003c2000, size 8192
Found memory based at 0x003c4000, size 4096
Found memory based at 0x003c5000, size 4096
```

17. Of course not! But the technique does actually work in some cases.

```
Found memory based at 0x003c6000, size 12288
Found memory based at 0x003c9000, size 4096
Found memory based at 0x003ca000, size 4096
Found memory based at 0x003cb000, size 4096
Found memory based at 0x003cc000, size 8192
Found memory based at 0x003e1000, size 12288
Found memory based at 0x003e5000, size 4096
Found memory based at 0x003f1000, size 24576
Found memory based at 0x003f8000, size 4096
Found memory based at 0x0042a000, size 8192
Found memory based at 0x0042c000, size 8192
Found memory based at 0x0042e000, size 8192
Found memory based at 0x00430000, size 4096
Found memory based at 0x00441000, size 491520
Found memory based at 0x004d8000, size 45056
Found memory based at 0x004f1000, size 20480
Found memory based at 0x004f7000, size 16384
Found memory based at 0x00500000, size 65536
Found memory based at 0x00700000, size 4096
Found memory based at 0x00790000, size 4096
Found memory based at 0x0089c000, size 4096
Found memory based at 0x0089d000, size 12288
Found memory based at 0x0099c000, size 4096
Found memory based at 0x0099d000, size 12288
Found memory based at 0x00a9e000, size 4096
Found memory based at 0x00a9f000, size 4096
Found memory based at 0x00aa0000, size 503808
Found memory based at 0x00c7e000, size 4096
Found memory based at 0x00c7f000, size 135168
Found memory based at 0x00cae000, size 4096
Found memory based at 0x00caf000, size 4096
Found memory based at 0x0ffed000, size 8192
Found memory based at 0x0ffef000, size 4096
Found memory based at 0x1001f000, size 4096
Found memory based at 0x10020000, size 12288
Found memory based at 0x10023000, size 4096
Found memory based at 0x10024000, size 4096
Found memory based at 0x71a83000, size 8192
Found memory based at 0x71a95000, size 4096
Found memory based at 0x71aa5000, size 4096
Found memory based at 0x71ac2000, size 4096
Found memory based at 0x77c58000, size 8192
Found memory based at 0x77c5a000, size 20480
Found memory based at 0x77cac000, size 4096
Found memory based at 0x77d2f000, size 4096
```

```
Found memory based at 0x77d9d000, size 8192
Found memory based at 0x77e36000, size 4096
Found memory based at 0x77e37000, size 8192
Found memory based at 0x77e39000, size 8192
Found memory based at 0x77ed6000, size 4096
Found memory based at 0x77ed7000, size 8192
Found memory based at 0x77fc5000, size 20480
Found memory based at 0x7ffd9000, size 4096
Found memory based at 0x7ffda000, size 4096
Found memory based at 0x7ffdb000, size 4096
Found memory based at 0x7ffdc000, size 4096
Found memory based at 0x7ffdd000, size 4096
Found memory based at 0x7ffde000, size 4096
Found memory based at 0x7ffdf000, size 4096
```

You can read all these memory sections and store them. You can think of this as a snapshot of the program. If you allow the program to continue executing, you can freeze it at any time in the future using another breakpoint. At any point where the program is frozen, you can then write back the original memory that you saved earlier. This effectively "restarts" the program at the point where you took the snapshot. This means you can continually keep "rewinding" the program in time.

For automated testing, this is a powerful technique. You can take a snapshot of a program and restart it. After restoring the memory you can then fiddle with memory, add corruption, or simulate different types of attack input. Then, once running, the program will act on the faulty input. You can apply this process in a loop and keep testing the same code with different perturbation of input. This automated approach is very powerful and can allow you to test millions of input combinations.

The following code illustrates how to take a snapshot of a target process. The code performs a query on the entire possible range of memory. For each valid location, the memory is copied into a list of structures:

```
struct mb
{
    MEMORY_BASIC_INFORMATION    mbi;
    char *p;
};

std: :list<struct mb *> gMemList;

void takesnap()
```

```
{
    DWORD start = 0;
    SIZE_T lpRead;

    while(start < 0xFFFFFFFF)
    {
        MEMORY_BASIC_INFORMATION mbi;

        int sz =
        VirtualQueryEx( hProcess,
                        (void *)start,
                        &mbi,
                        sizeof(MEMORY_BASIC_INFORMATION));

        if(    (mbi.State == MEM_COMMIT)
               &&
               (mbi.Protect != PAGE_READONLY)
               &&
               (mbi.Protect != PAGE_EXECUTE_READ)
               &&
               (mbi.Protect != PAGE_GUARD)
               &&
               (mbi.Protect != PAGE_NOACCESS)
               )
        {
            TRACE("Found memory based at %d, size %d\n",
                   mbi.BaseAddress,
                   mbi.RegionSize);
            struct mb *b = new mb;
            memcpy(    (void *)&(b->mbi),
                   (void *)&mbi,
                   sizeof(MEMORY_BASIC_INFORMATION));

            char *p = (char *)malloc(mbi.RegionSize);
            b->p = p;

            if(!ReadProcessMemory( hProcess,
                                   (void *)start, p,
                                   mbi.RegionSize, &lpRead))
            {
                   TRACE("ReadProcessMemory failed %d\nRead %d",
                   GetLastError(), lpRead);
            }
            if(mbi.RegionSize != lpRead)
            {
```

```
                    TRACE("Read short bytes %d != %d\n",
                    mbi.RegionSize,
                    lpRead);
            }
            gMemList.push_front(b);
        }

        if(start + mbi.RegionSize < start) break;
        start += mbi.RegionSize;
    }
}
```

The code uses the `VirtualQueryEx` API call to test each location of memory from `0` to `0xFFFFFFFF`. If a valid memory address is found, the size of the memory region is obtained and the next query is placed just beyond the current region. In this way the same memory region is not queried more than once. If the memory region is committed, then this means it's being used. We check that the memory is not read-only so that we only save memory regions that might be modified. Clearly, read-only memory is not going to be modified, so there is no reason to save it. If you are really careful, you can save all the memory regions. You may suspect that the target program changes the memory protections during execution, for example.

If you want to restore the program state, you can write back all the saved memory regions:

```
void setsnap()
{
    std::list<struct mb *>::iterator ff = gMemList.begin();
    while(ff != gMemList.end())
    {
        struct mb *u = *ff;
        if(u)
        {
            DWORD lpBytes;
            TRACE("Writing memory based at %d, size %d\n",
                    u->mbi.BaseAddress,
                    u->mbi.RegionSize);

            if(!WriteProcessMemory(hProcess,
                                   u->mbi.BaseAddress,
                                   u->p,
                                   u->mbi.RegionSize,
```

```
                                    &lpBytes))
            {
                    TRACE("WriteProcessMemory failed, error %d\n",
                    GetLastError());
            }
            if(lpBytes != u->mbi.RegionSize)
            {
                    TRACE("Warning, write failed %d != %d\n",
                            lpBytes,
                            u->mbi.RegionSize);
            }
        }
        ff++;
    }
}
```

The code to write back the memory is much simpler. It does not need to query the memory regions; it simply writes the memory regions back to their original locations.

Disassembling Machine Code

A debugger needs to be able to disassemble instructions. A breakpoint or single-step event will leave each thread of the target process pointing to some instruction. By using the thread `CONTEXT` functions you can determine the address in memory where the instruction lives, but this does not reveal the actual instruction itself.

The memory needs to be "disassembled" to determine the instruction. Fortunately you don't need to write a disassembler from scratch. Microsoft supplies a disassembler with the OS. This disassembler is used, for example, by the Dr. Watson utility when a crash occurs. We can borrow from this existing tool to provide disassembly functions in our debugger:

```
HANDLE hThread =
fOpenThread(
              THREAD_ALL_ACCESS,
              FALSE,
              theThread->m_thread_id
              );

if(hThread == NULL)
{
    _error_out("[!] Failed to Open the thread handle !\n");
```

```
        return FALSE;
    }

    DEBUGPACKET dp;
    dp.context = theThread->m_ctx;
    dp.hProcess = theThread->m_hProcess;
    dp.hThread = hThread;

    DWORD ulOffset = dp.context.Eip;

    // Disassemble the instruction.
    if ( disasm ( &dp        ,
                  &ulOffset     ,
                  (PUCHAR)m_instruction,
                  FALSE       ) )
    {
        ret = TRUE;
    }
    else
    {
        _error_out("error disassembling instruction\n");
        ret = FALSE;
    }

    CloseHandle(hThread);
```

A user-defined thread structure is used in this code. The context is obtained so we know which instruction is being executed. The `disasm` function call is published in the Dr. Watson source code and can easily be incorporated into your project. We encourage you to locate the source code to Dr. Watson to add the relevant disassembly functionality. Alternatively, there are other open-source disassemblers available that provide similar functionality.

Building a Basic Code Coverage Tool

As we mentioned early in the chapter, all the available coverage tools, commercial or otherwise, lack significant features and data visualization methods that are important to the attacker. Instead of fighting with expensive and deficient tools, why not write your own? In this section we present one of the jewels of this book—a simple code coverage tool that can be designed using the debugging API calls that are described elsewhere in this book. The tool should track all conditional branches in the code. If the

conditional branch can be controlled by user-supplied input, this should be noted. Of course, the goal is to determine whether the input set has exercised all possible branches that can be controlled.

For the purposes of this example, the tool will run the processor in single-step mode and will track each instruction using a disassembler. The core object we are tracking is a code *location*. A location is a single continuous block of instructions with no branches. Branch instructions connect all the code locations together. That is, one code location branches to another code location. We want to track all the code locations that have been visited and determine whether user-supplied input is being processed in the code location. The structure we are using to track code locations is as follows:

```
// A code location
struct item
{
    item()
    {
        subroutine=FALSE;
        is_conditional=FALSE;
        isret=FALSE;
        boron=FALSE;
        address=0;
        length=1;
        x=0;
        y=0;
        column=0;
        m_hasdrawn=FALSE;
    }

    bool    subroutine;
    bool    is_conditional;
    bool    isret;
    bool    boron;
    bool    m_hasdrawn;        // To stop circular references

    int        address;
    int        length;
    int        column;
    int        x;
    int        y;

    std::string m_disasm;
```

```
    std::string m_borons;

    std::list<struct item *> mChildren;

    struct item * lookup(DWORD addr)
    {
        std::list<item *>::iterator i = mChildren.begin();
        while(i != mChildren.end())
        {
            struct item *g = *i;
            if(g->address == addr) return g;
            i++;
        }
        return NULL;
    }
};
```

Each location has a list of pointers to all branch targets from the location. It also has a string that represents the assembly instructions that make up the location. The following code executes on each single-step event:

```
struct item *anItem = NULL;

// Make sure we have a fresh context.
theThread->GetThreadContext();

// Disassemble the target instruction.
m_disasm.Disasm( theThread );

// Determine if this is the target of a branch instruction.
if(m_next_is_target || m_next_is_calltarget)
{
    anItem = OnBranchTarget( theThread );
    SetCurrentItemForThread( theThread->m_thread_id, anItem);
    m_next_is_target = FALSE;
    m_next_is_calltarget = FALSE;

    // We have branched, so we need to set the parent/child
    // lists.
    if(old_item)
    {
        // Determine if we are already in the child.
        if(NULL == old_item->lookup(anItem->address))
```

```
            {
                old_item->mChildren.push_back(anItem);
            }
        }
    }
    else
    {
        anItem = GetCurrentItemForThread(
    theThread->m_thread_id );
    }

    if(anItem)
    {
        anItem->m_disasm += m_disasm.m_instruction;
        anItem->m_disasm += '\n';
    }
    char *_c = m_disasm.m_instruction;
    if(strstr(_c, "call"))
    {
        m_next_is_calltarget = TRUE;
    }
    else if(strstr(_c, "ret"))
    {
        m_next_is_target = TRUE;
        if(anItem) anItem->isret = TRUE;
    }
    else if(strstr(_c, "jmp"))
    {
        m_next_is_target = TRUE;
    }
    else if(strstr(_c, "je"))
    {
        m_next_is_target = TRUE;
        if(anItem)anItem->is_conditional=TRUE;
    }
    else if(strstr(_c, "jne"))
    {
        m_next_is_target = TRUE;
        if(anItem)anItem->is_conditional=TRUE;
    }
    else if(strstr(_c, "jl"))
    {
        m_next_is_target = TRUE;
        if(anItem)anItem->is_conditional=TRUE;
    }
```

```
    else if(strstr(_c, "jle"))
    {
        m_next_is_target = TRUE;
        if(anItem)anItem->is_conditional=TRUE;
    }
    else if(strstr(_c, "jz"))
    {
        m_next_is_target = TRUE;
        if(anItem)anItem->is_conditional=TRUE;
    }
    else if(strstr(_c, "jnz"))
    {
        m_next_is_target = TRUE;
        if(anItem)anItem->is_conditional=TRUE;
    }
    else if(strstr(_c, "jg"))
    {
        m_next_is_target = TRUE;
        if(anItem)anItem->is_conditional=TRUE;
    }
    else if(strstr(_c, "jge"))
    {
        m_next_is_target = TRUE;
        if(anItem)anItem->is_conditional=TRUE;
    }
    else
    {
        // Not a branching instruction,
        // so add one to the current item length.
        if(anItem) anItem->length++;
    }

    ////////////////////////////////////////////
    // Check for boron tag.
    ////////////////////////////////////////////
    if(anItem && mTagLen)
    {
        if(check_boron(theThread, _c, anItem)) anItem->boron = TRUE;
    }

    old_item = anItem;
```

First, we see the code gets a fresh context structure for the thread that just single stepped. The instruction pointed to by the instruction pointer is

disassembled. If the instruction is the beginning of a new code location, the list of currently mapped locations is queried so that we don't make double entries. The instruction is then compared with a list of known branching instructions, and appropriate flags are set in the item structure. Finally, a check is made for boron tags. The code for a boron tag check is presented in the following paragraph.

Checking for Boron Tags

When a breakpoint or single-step event has occurred, the debugger may wish to query memory for boron tags (that is, substrings that are known to be user supplied). Using the memory query routines introduced earlier in the book, we can make some fairly intelligent queries for boron tags. Because CPU registers are used constantly to store pointers to data, it makes sense to check all the CPU registers for valid memory pointers when the breakpoint or single step has occurred. If the register points to valid memory, we can then query that memory and look for a boron tag. The fact is that any code location that is using user-supplied data typically has a pointer to these data in one of the registers. To check the registers, you can use a routine like this:

```
bool check_boron( CDThread *theThread, char *c, struct item *ip )
{
    // If any of the registers point to the user buffer, tag this.
    DWORD reg;

    if(strstr(c, "eax"))
    {
        reg = theThread->m_ctx.Eax;
        if(can_read( theThread, (void *)reg ))
        {
            SIZE_T lpRead;
            char string[255];
            string[mTagLen]=NULL;
            // Read the target memory.
            if(ReadProcessMemory( theThread->m_hProcess,
                    (void *)reg, string, mTagLen, &lpRead))
            {
                    if(strstr( string, mBoronTag ))
                {
                    // Found the boron string.
                    ip->m_borons += "EAX: ";
                    ip->m_borons += c;
                    ip->m_borons += " -> ";
```

```
                    ip->m_borons += string;
                    ip->m_borons += '\n';

                    return TRUE;
                }
            }
        }
    }
....
// Repeat this call for all the registers EAX, EBX, ECX, EDX, ESI, and EDI.

return FALSE;
}pl;
```

To save room, we didn't paste the code for all registers, just the EAX register. The code should query all registers listed in the comment. The function returns `TRUE` if the supplied `boron` tag is found behind one of the memory pointers.

Conclusion

All software is made up of machine-readable code. In fact, code is what makes every program function the way it does. The code defines the software and the decisions it will make. Reverse engineering, as applied to software, is the process of looking for patterns in this code. By identifying certain code patterns, an attacker can locate potential software vulnerabilities.

This chapter has exposed you to the basic concepts and methods of decompilation, all in the name of better understanding how a program really works. We've even gone so far as to provide some rudimentary (yet still powerful) tools as examples. Using these methods and tools, you can learn almost anything you need to know about a target, and then use this information to exploit it.

Praise for *Exploiting Software*

"*Exploiting Software* highlights the most critical part of the software quality problem. As it turns out, software quality problems are a major contributing factor to computer security problems. Increasingly, companies large and small depend on software to run their businesses every day. The current approach to software quality and security taken by software companies, system integrators, and internal development organizations is like driving a car on a rainy day with worn-out tires and no air bags. In both cases, the odds are that something bad is going to happen, and there is no protection for the occupant/owner.

This book will help the reader understand how to make software quality part of the design—a key change from where we are today!"

Tony Scott
Chief Technology Officer, IS&S
General Motors Corporation

"It's about time someone wrote a book to teach the good guys what the bad guys already know. As the computer security industry matures, books like *Exploiting Software* have a critical role to play."

Bruce Schneier
Chief Technology Officer
Counterpane
Author of Beyond Fear *and* Secrets and Lies

"*Exploiting Software* cuts to the heart of the computer security problem, showing why broken software presents a clear and present danger. Getting past the 'worm of the day' phenomenon requires that someone other than the bad guys understands how software is attacked.

This book is a wake-up call for computer security."

Elinor Mills Abreu
Reuters' correspondent

"Police investigators study how criminals think and act. Military strategists learn about the enemy's tactics, as well as their weapons and personnel capabilities. Similarly, information security professionals need to study their criminals and enemies, so we can tell the difference between popguns and weapons of mass destruction. This book is a significant advance in helping the 'white hats' understand how the 'black hats' operate.

Through extensive examples and 'attack patterns,' this book helps the reader understand how attackers analyze software and use the results of the analysis to attack systems. Hoglund and McGraw explain not only how hackers attack servers, but also how malicious server operators can attack clients (and how each can protect themselves from the other). An excellent book for practicing security engineers, and an ideal book for an undergraduate class in software security."

Jeremy Epstein
Director, Product Security & Performance
webMethods, Inc.

"A provocative and revealing book from two leading security experts and world class software exploiters, *Exploiting Software* enters the mind of the cleverest and wickedest crackers and shows you how they think. It illustrates general principles for breaking software, and provides you a whirlwind tour of techniques for finding and exploiting software vulnerabilities, along with detailed examples from real software exploits.

Exploiting Software is essential reading for anyone responsible for placing software in a hostile environment—that is, everyone who writes or installs programs that run on the Internet."

Dave Evans, Ph.D.
Associate Professor of Computer Science
University of Virginia

"The root cause for most of today's Internet hacker exploits and malicious software outbreaks are buggy software and faulty security software deployment. In *Exploiting Software,* Greg Hoglund and Gary McGraw help us in an interesting and provocative way

to better defend ourselves against malicious hacker attacks on those software loopholes.

The information in this book is an essential reference that needs to be understood, digested, and aggressively addressed by IT and Information Security professionals everywhere."

Ken Cutler, CISSP, CISA
Vice President, Curriculum Development & Professional Services,
MIS Training Institute

"This book describes the threats to software in concrete, understandable, and frightening detail. It also discusses how to find these problems before the bad folks do. A valuable addition to every programmer's and security person's library!"

Matt Bishop, Ph.D.
Professor of Computer Science
University of California at Davis
Author of Computer Security: Art and Science

"Whether we slept through software engineering classes or paid attention, those of us who build things remain responsible for achieving meaningful and measurable vulnerability reductions. If you can't afford to stop all software manufacturing to teach your engineers how to build secure software from the ground up, you should at least increase awareness in your organization by demanding that they read *Exploiting Software*. This book clearly demonstrates what happens to broken software in the wild."

Ron Moritz, CISSP
Senior Vice President, Chief Security Strategist
Computer Associates

"*Exploiting Software* is the most up-to-date technical treatment of software security I have seen. If you worry about software and application vulnerability, *Exploiting Software* is a must read. This book gets at all the timely and important issues surrounding software security in a technical, but still highly readable and engaging, way.

Hoglund and McGraw have done an excellent job of picking out the major ideas in software exploit and nicely organizing them to make sense of the software security jungle."

George Cybenko, Ph.D.
Dorothy and Walter Gramm Professor of Engineering, Dartmouth
Founding Editor-in-Chief, IEEE Security and Privacy

"This is a seductive book. It starts with a simple story, telling about hacks and cracks. It draws you in with anecdotes, but builds from there. In a few chapters you find yourself deep in the intimate details of software security. It is the rare technical book that is a readable and enjoyable primer but has the substance to remain on your shelf as a reference. Wonderful stuff."

Craig Miller, Ph.D.
Chief Technology Officer for North America
Dimension Data

"It's hard to protect yourself if you don't know what you're up against. This book has the details you need to know about how attackers find software holes and exploit them—details that will help you secure your own systems."

Ed Felten, Ph.D.
Professor of Computer Science
Princeton University