Vulnerability Discovery

Now that you are an expert at hacking Linux, Windows, Solaris, OS X, and Cisco, we move into the entire section of the book dedicated to discovering vulnerabilities. We cover the most popular methods used by hackers in the real world. First things first, you must set up a working environment, a platform to orchestrate vulnerability discovery from. In Chapter 15 we cover the tools and reference material you will need for productive and efficient vulnerability discovery. Chapter 16 introduces one of the more popular methods of automated vulnerability discovery, fault injection. A similar method of automated bug finding is detailed in Chapter 17, fuzzers.

Other forms of vulnerability discovery are just as valid as fuzzing, so they are covered as well. Discovering vulnerabilities by auditing source code is important, as more and more important applications come with source code; Chapter 18 describes this method of bug hunting when you have source code. Manual methods of vulnerability discovery have proven to be highly successful, so Chapter 19 goes over instrumented investigation, using tried and true techniques for finding security bugs manually. Chapter 20 covers vulnerability tracing, a method of tracing where input is copied through many different functions, modules, and libraries. Finally, auditing binaries in Chapter 21 rounds out this part with a comprehensive tutorial on discovering vulnerabilities when you have only a binary to work with.

CHAPTER 15

Establishing a Working Environment

If you exploit overflows and format strings and other shellcode-level issues, you need a good working environment. By *environment*, I don't mean a darkened room with a lot of pizza and diet soda. I refer to a good set of coding tools, tracing tools, and reference materials that will help you accomplish your tasks with minimum fuss. This chapter will give you a starting point to establish that environment.

Generally speaking, if you want to exploit a bug, you need at least two items: a set of reference papers and manuals that give you the information you need about the system you're exploiting and a set of coding tools so that you can write the exploit. In addition, a set of tools you can use for *tracing* (closely observing the system under test) is very useful. We'll start by giving you a quick overview of the more popular items in each of these three categories. Because something new comes along in the shellcode world pretty much on a daily basis, don't take this as a cutting-edge, state-of-the-art discussion of what's out there; rather, it's a quick compendium of the very best references, coding tools, and tracing tools available at time of writing.

Also, we do not favor a specific OS, so not all the items listed will relate to the OS you're targeting. I list the relevant OS if it is important—if no OS is listed, then either the item is a tool that runs pretty much on everything, or it is a paper that applies to a general class of problem.

What You Need for Reference

First, you'll need some assembler references for the target architecture:

- Intel x86
- *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*
  `http://www.intel.com/design/mobile/manuals/243191.htm`
  Or do an Internet search for 24319101.pdf
- X86 Assembly Language FAQ
  `http://www.faqs.org/faqs/assembly-language/x86/`
- IA64 references (Itanium)
  `http://www.intel.com/design/itanium/manuals/iiasdmanual.htm`
- SPARC Assembly Language Reference Manual
  `http://docs.sun.com/app/docs/doc/816-1681`
  Or do an Internet search for 816-1681.pdf
- SPARC Architecture Online Reference Manual
  `http://online.mq.edu.au/pub/COMP226/sparc-manual/index.html`
- PA/RISC reference manuals (HP)
  Search for "References and Manuals" on the HP site
- `http://lsd-pl.net/` contains a good collection of papers.

What You Need for Code

In order to write your code, you'll need some tools. What follows is a brief discussion of some of the more popular tools among x86 shellcoders.

gcc

gcc (*GNU Compiler Collection*) is actually much more than a C/C++ compiler; gcc also contains front ends for Fortran, Java, and Ada. It is almost certainly the best free (GPL) compiler available, and with its support for inline assembly, it is an excellent choice for the shellcode developer.

The gcc home page is `http://gcc.gnu.org/`.

gdb

gdb (*GNU Debugger*) is a free (GPL) debugger that integrates well with gcc and provides a command-line-based symbolic debugging environment. It also has

excellent support for interactive disassembly and is thus a good choice for investigating the initial vectors for an overflow/format string bug.

You can find gdb at `http://www.gnu.org/software/gdb/`.

NASM

NASM (*Netwide Assembler*) is a free x86 assembler supporting a variety of output binary file formats, such as Linux and BSD a.out, ELF, COFF, and 16- and 32-bit Windows object and executable formats.

NASM is an extremely useful tool if you need a dedicated assembler. It also has an excellent x86 opcode reference in its documentation.

You can find NASM at `http://sourceforge.net/projects/nasm`.

WinDbg

WinDbg is a standalone debugger for the Windows platform supplied by Microsoft. It features a friendly GUI interface with a number of excellent features, including memory searching, the ability to debug child processes, and extensive exception handling facilities. WinDbg is useful if you want to write an exploit for a program on the Windows platform that starts child processes (such as Oracle or Apache), because it can automatically follow and attach to them.

You can find WinDbg at `http://www.microsoft.com/whdc/devtools/debugging/`, or via an Internet search for *Debugging tools for Windows*.

OllyDbg

OllyDbg is a Windows "analyzing debugger." OllyDbg contains extremely nice features such as a full memory search (WinDbg lacks this) and a great disassembler. Using OllyDbg is much like having most of the best parts of WinDbg and IDA in a single, free tool.

You can find OllyDbg at `http://www.ollydbg.de/`.

Visual C++

Visual C++ is Microsoft's flagship C/C++ compiler. It has an excellent user interface, and full debugging facilities are built in. Visual C++ integrates fully with the Microsoft Developer Network documentation set (MSDN), which can be extremely useful if you're writing Windows exploits—having a good Win32 API reference integrated into your IDE makes things much quicker. Like gcc, Visual C++ supports inline assembly, which makes exploit development simpler. All in all, if you have access to a license for Visual C++/Developer Studio, it's worth a look.

Python

Lately, many exploit coders have been writing their exploits in Python, a language well known for rapid application development. Two of the authors of this book, for example, use Python to gain a competitive advantage in the world of rapid and effective exploit development. With the addition of MOSDEF, a pure Python assembler and shellcode development tool, Python can be one of the most effective tools in your arsenal.

What You Need for Investigation

In order to find bugs, you'll need a good idea of what's going on in the internal structures of the application or program you are attacking. The tools listed in this section are useful in a variety of situations, such as when you're hunting for bugs, when you're developing an exploit, and when you're trying to work out what someone else's exploit does.

Useful Custom Scripts/Tools

In addition to the tools listed in this chapter, the authors use a variety of small, custom tools for various purposes. You might want to write your own scripts or tools for similar purposes.

An Offset Finder

On both Windows and Unix platforms, you will frequently need to find the address of a given instruction. For example, in a Windows stack overflow you might find that the `ESP` register is pointing at your shellcode. In order to exploit this, you'll need to find the address of some instruction stream that redirects execution to your code. The easiest way to do this is to find one of the following byte sequences in memory and then overwrite the saved return address with the address of one of the following sequences:

```
jmp esp             (0xff 0xe4)
call esp            (0xff 0xd4)
push esp; ret       (0x54 0xc3)
```

You should find these sequences in numerous places in memory. Ideally, you should look for them in DLLs that haven't changed across service packs. An offset finder typically works by attaching to the remote process and suspending all its threads, and then hunting through memory for the specified byte sequences, reporting them into a text file. It's a simple, but useful, thing to

have. Alternatively, the Metasploit project has an opcode database online at `http://www.metasploit.com/opcode_database.html`.

Generic Fuzzers

If you're investigating a given product for security flaws, you'll probably find it useful to write a fuzzer that focuses on some specific feature of the product—a Web interface or some custom network protocol or maybe even an RPC interface. Again, generic fuzzers are useful things to have around. Even very simple fuzzers can accomplish much.

The Debug Trick

Reverse shells in Windows can be pretty frustrating. You can't easily upload files, and the basic scripting support is limited. There is a ray of hope in this dank and frightening world, however, which comes in the unlikely form of the old MS-DOS debugger, `debug.exe`.

You will find `debug.exe` on nearly every Windows box. It's been around since MS-DOS, and it still exists in the latest release of Windows XP. Although `debug.exe` was primarily intended as a tool to debug and create `.com` files, you can also use it to create an arbitrary binary file—with certain limits. The file must be less than 64K, and the filename cannot end in `.exe` or `.com`.

For example, take the following binary file:

```
73 71 75 65 61 6D 69 73 68 20 6F 73 73 69 66 72    squeamish ossifr
61 67 65 0A DE C0 DE DE C0 DE DE C0 DE    age.@@pÀ@@p@@pÀ@@p@@pÀ@@p
```

You can write a script file that outputs said binary file, as follows (call it `foo.scr`):

```
n foo.scr
e 0000  73 71 75 65 61 6d 69 73 68 20 6f 73 73 69 66 72
e 0010  61 67 65 0d 0a de c0 de de c0 de de c0 de
rcx
1e
w 0
q
```

Then run, `debug.exe`.

```
debug < foo.scr
```

`debug.exe` will output the binary file.

The point here is that the script file needs only to contain alphanumeric characters, so that you can use the `echo` command over a reverse shell to create it.

Once the script file is on the remote host, you run `debug.exe` in the manner just specified, and bingo, you have your binary file. You can simply rename the initial file, for example, `nc.foo`, and then (once you've uploaded it) rename it `nc.exe`.

The only thing you must automate is the creation of the script file. Once again, this is easily done in perl, Python, or C. `debug.exe` is an exceptionally useful tool if you insist on using reverse shells in Windows.

There are other ways of achieving a binary-upload on Windows—for instance, it's possible to create a `.com` file consisting purely of printable characters that can be used to create an arbitrary binary file. You "echo" the `.com` file and then repeatedly call it to create the target file.

All Platforms

Probably the single-most popular network security tool in existence that can be used on all platforms is NetCat. Its original author, Hobbit, described NetCat as his "TCP/IP Swiss army knife." NetCat allows you to send and receive arbitrary data on arbitrary TCP and UDP ports, as well as listen for (for example) reverse shells. NetCat ships with quite a few Linux distributions as standard, and it has a Windows port. There's even a GNU version at `http://netcat.sourceforge.net/`.

The original Unix and Windows versions—by Hobbit and Chris Wysopal (Weld Pond) can be found at `http://www.vulnwatch.org/netcat/`.

Unix

In general, it's easier to see what's going on in a Unix system than it is in a Windows system; therefore, bug hunters have a slightly easier time of it.

ltrace and strace

ltrace and strace are programs that allow you to view the dynamic library calls and syscalls that a program makes, as well as view the signals the program receives. ltrace is exceptionally useful if you're trying to work out how a particular part of a certain string-handling mechanism works in a target process. strace is also pretty useful if you're trying to evade a host-based IDS and need to work out what pattern of syscalls a program makes.

For more information, simply check the man pages for *ltrace* and *strace*.

truss

truss provides much the same functionality as a combined ltrace and strace on Solaris.

fstat (BSD)

fstat is a BSD-based utility for identifying open files (including sockets). It's pretty useful for quickly seeing which processes are doing what in a complex environment.

tcpdump

Because the best bugs are remote bugs, a packet sniffer is essential. tcpdump can be useful for obtaining a quick overview of what a particular daemon is doing; for more detailed analysis, however, Wireshark (discussed next) is probably better.

Wireshark (formerly Ethereal)

Wireshark is a GUI-based free network packet sniffer and analyzer. It has a huge number packet parsers, so it's a pretty good first choice if you're trying to understand an unusual network protocol or if you're writing a protocol fuzzer.

You can download Wireshark at `www.wireshark.org/`.

Windows

A bug hunter's life on the Windows platform is a slightly harder one. The following tools are exceptionally useful—all of them can be downloaded from Mark Russinovich and Bryce Cogswell's excellent Sysinternals site, now moved to Microsoft, at `http://www.microsoft.com/technet/sysinternals/default.mspx`.

- **RegMon**—Monitors access to the Windows registry, with a filter so that you can focus on the processes under test.
- **FileMon—**Monitors file activity, again with a useful filter feature.
- **HandleEx**—Views DLLs loaded by a process, as well as all the handles it has open; for example, named pipes, shared memory sections, and files.
- **TCPView**—Associates TCP and UDP endpoints with the process that owns them.
- **Process Explorer**—Allows real-time examination of processes, handles, DLLs, and more

The Sysinternals site provides many excellent tools, but these five programs make up a good starting toolkit.

IDA Pro Disassembler

The IDA pro disassembler is *the* best disassembly tool on the market for the Windows reverse engineer. It features an excellent, scriptable user interface with easy cross referencing and search facilities. IDA Pro is especially useful when you need to establish exactly what certain vulnerable code is doing and when you're having trouble with such tasks as continuation of execution or socket stealing. You can find IDA at `www.datarescue.com/`.

What You Need to Know

Numerous papers exist on how to write exploits for stack overflows; there are slightly fewer about format strings, and still fewer about heap overflows. If the bug you're trying to exploit is not one of these three, then you probably will have difficulty obtaining the relevant information. Hopefully this book fills in many of the gaps, but if you need more information on a certain bug, the following list might help. We deliberately kept this list of our favorite papers in each category brief.

Keep in mind that reading old exploits can be just as valuable as reading papers. Often, the comments and headers detail particular techniques that may be of interest to novice exploit developers.

There is much excellent information out there that we've had to omit for sake of space, so please accept our apologies if your own paper is not listed.

Stack Overflow Basics

- "Smashing the Stack for Fun and Profit" (Aleph One)
  Phrack Magazine, issue 49, article 14
  `http://www.phrack.org/archives/49/P49-14`
- Exploiting Windows NT 4 Buffer Overruns (David Litchfield)
  `www.ngssoftware.com/papers/ntbufferoverflow.html`
- "Win32 Buffer Overflows: Location, Exploitation and Prevention"
  (dark spyrit, Barnaby Jack, `dspyrit@beavuh.org`)
  Phrack Magazine, issue 55, article 15
  `http://www.phrack.org/archives/55/P55-15`
- The Art of Writing Shellcode (smiler)
  `http://julianor.tripod.com/art-shellcode.txt`
- The Tao of Windows Buffer Overflow
  (as taught by DilDog)
  `www.cultdeadcow.com/cDc_files/cDc-351/`

- Unix Assembly Codes Development for Vulnerabilities Illustration Purposes (LSD-PL)
  `http://lsd-pl.net/projects/asmcodes.zip`

Advanced Stack Overflows

- Using Environment for Returning into Lib C (Lupin Bursztein)
  `www.shellcode.com.ar/docz/bof/rilc.html` (Lupin's home page is `www.bursztein.net`; however, the paper was not there at time of writing)
- Non-Stack Based Exploitation of Buffer Overrun Vulnerabilities on Windows NT/2000/XP (David Litchfield)
  `www.ngssoftware.com/papers/non-stack-bo-windows.pdf`
- Bypassing Stackguard and StackShield Protection (Gerardo Richarte)
  `www.coresecurity.com/common/showdoc.php?idx=242&idxseccion=11`
- Vivisection of an Exploit (Dave Aitel)
  Blackhat Briefings Presentation, Amsterdam 2003
  `www.blackhat.com/presentations/bh-europe-03/bh-europe-03-aitel.pdf`

Heap Overflow Basics

- w00w00 on Heap Overflows (Matt Conover)
  `www.w00w00.org/files/articles/heaptut.txt`
- "Once upon a free()"
  Phrack Magazine, Issue 57, Article 9
  `http://www.phrack.org/archives/57/p57-0x09`
- "Vudo—An object superstitiously believed to embody magical powers" (Michel MaXX Kaempf, `maxx@synnergy.net`)
  Phrack Magazine, Issue 57, Article 8
  `http://www.phrack.org/archives/57/p57-0x08`

Integer Overflow Basics

- "Basic Integer Overflows" (blexim)
  Phrack Magazine, Issue 60, Article 10
  `http://www.phrack.org/archives/60/p60-0x0a.txt`

Format String Basics

- Format String Attacks (Tim Newsham)
  `http://community.corest.com/~juliano/tn-usfs.pdf`

- Exploiting Format String Vulnerabilities (scut)
  `http://julianor.tripod.com/teso-fs1-1.pdf`
- "Advances in Format String Exploitation" (Gera, Riq)
  Phrack Magazine, Issue 59, Article 7
  `http://www.phrack.org/archives/59/p59-0x07.txt`

Encoders and alternatives

- "Writing ia32 Alphanumeric Shellcodes" (rix)
  Phrack Magazine, Issue 57, Article 15
  `http://www.phrack.org/archives/57/p57-0x18`
- Creating Arbitrary Shellcode in Unicode Expanded Strings (Chris Anley)
  `http://www.ngssoftware.com/papers/unicodebo.pdf`

Tracing, Bugging, and Logging

- The VTrace Tool: Building a System Tracer for Windows NT and Windows 2000
  "VTrace" system tracing tool (explanatory article)
  `http://msdn.microsoft.com/msdnmag/issues/1000/VTrace/`
- "Interception of Win32 API Calls" (MS Research Paper)
  `www.research.microsoft.com/sn/detours/`
- "Writing [a] Linux Kernel Keylogger" (rd)
  Phrack Magazine, Issue 59, Article 14
  `http://www.phrack.org/archives/59/p59-0x17`
- "Hacking the Linux Kernel Network Stack" (bioforge)
  Phrack Magazine, Issue 61, Article 13
  `http://www.phrack.org/archives/61/p61-0x0d_Hacking_the_Linux_Kernel_Network_Stack.txt`
- "Analysis: .ida 'Code Red' Worm" (Ryan Permeh, Marc Maiffret)
  `www.eeye.com/html/Research/Advisories/AL20010717.html`

Paper Archives

The following list contains archives of useful papers. Most of these archives link to many of the papers previously listed, as well as to other useful texts.

- `http://community.corest.com/~juliano/`
- `http://packetstormsecurity.nl/papers/unix/`

Optimizing Shellcode Development

The first exploit you write will be the most difficult and tedious. As you accumulate more exploits and more experience, you will learn to optimize various tasks in order to reduce the time between finding a bug and obtaining your nicely packaged exploit. This section is a brief attempt to distill our techniques into a short, readable guide to optimizing the development process.

Of course, the best way to speed shellcode development is to not actually write the shellcode—use a syscall proxy or proglet mechanism instead. Most of the time, however, a simple static exploit is the easiest thing to do, so let's talk about how to optimize that and improve its quality.

Plan the Exploit

Before you rush blindly into writing an exploit, it is a good idea to have a firm plan of the steps you'll take to exploit the bug. In the case of a vanilla stack overflow on the Windows platform, the plan might look like the following (depending on how you personally would write this kind of exploit):

1. Determine offset of bytes that overwrite saved return address.
2. Determine location of payload relative to registers. (Is `ESP` pointing at our buffer? Any other registers?)
3. Find a reliable `jmp/call <register>` offset for (a) the product version or (b) the various Windows versions and service packs you're targeting.
4. Create small test shellcode of `nops` to establish whether corruption is taking place.
5. If there is corruption, insert `jmps` into payload to avoid corrupted areas. If there is no corruption, substitute actual shellcode.

Write the Shellcode in Inline Assembler

This trick can save you a large amount of time. Most published exploits contain incomprehensible streams of hexadecimal bytes encoded in C string constants. This doesn't help if you need to insert a `jmp` to avoid a corrupt part of the stack or if you want to make a quick modification to your shellcode. Instead of C constants, try something like the following (this code is for Visual C++, but a similar technique works for gcc):

```
char *sploit()
{
      asm
```

```
    {
          ; this code returns the address of the start of the code
          jmp get_sploit
get_sploit_fn:
          pop eax
          jmp got_sploit
get_sploit:
          call get_sploit_fn ; get the current address into eax

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Exploit
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


; start of exploit

          jmp get_eip

get_eip_fn:

          pop edx
          jmp got_eip

get_eip:

          call get_eip_fn   ; get the current address into edx

call_get_proc_address:

          mov ebx, 0x01475533     ; handle for loadlibrary
          sub ebx, 0x01010101
          mov ecx, dword ptr [ebx]
```

And so on. Writing code this way has several advantages:

- You can comment the assembler code easily, which helps when you need to modify your shellcode six months later.
- You can debug the shellcode and test it, with comments and easy breakpointing, without actually firing off the exploit. Breakpointing is useful if your exploit does more than just spawn a shell.
- You can easily cut and paste sections of shellcode from other exploits.
- You don't need to go through an arcane cut-and-pasting exercise every time you want to change the code—simply change the assembler and run it.

Of course, writing the exploit this way changes the harness slightly, so you need a method for determining the length of the exploit. One method is to

avoid using instructions that result in null bytes, and then paste instructions at the end of the shellcode.

```
add         byte ptr [eax],al
```

Remember, the preceding assembles to two null bytes. The harness can then simply do a `strlen` to find the exploit length.

Maintain a Shellcode Library

The quickest way to write code is to cut and paste from code that already works. If you're writing an exploit, it doesn't matter so much whether the code you're cutting is your own or someone else's, as long as you understand exactly what it's doing. If you don't understand what a piece of shellcode is doing, it is probably quicker in the long run to write something to perform that task yourself, because you'll then be able to modify it more easily.

Once you have a few working exploits, you will tend to settle into using the same generic payloads, but it is always useful to have other, more complex codes nearby for easy reference. Simply put code snippets into some easily searchable form, such as a hierarchy of directories containing text files. A quick `grep` and you've got the code you need.

Make It Continue Nicely

Continuation of execution is an exceptionally complex subject, but it's key to writing high-quality exploits. Here is a quick list of approaches to the problem, along with other useful information:

- If you terminate the target process, does it get restarted? If so, call `exit()` or `ExitProcess()` or `TerminateProcess()` in Windows.
- If you terminate the target thread, does it get restarted? If so, call `ExitThread()`, `TerminateThread()`, or the equivalent. This method works extremely well if you're exploiting a DBMS, because they tend to use pools of worker threads. (Oracle and SQL Server both do this.)
- If you have a heap overflow, can you repair the heap? This is kind of tricky, but this book lists some good pointers.

In terms of restoring the flow of control, you have a few alternatives:

- **Trigger an exception handler.** Check for exception handlers first on the general principle that the easiest code to write is the code you don't write. If the target process already has a full-featured exception handler that cleans up nicely and restarts everything, why not just call it, or trigger it by causing an exception?

- **Repair the stack and return to parent.** This technique is tricky, because there's probably information on the stack that you can't easily obtain by searching memory. However, this method is possible in some cases. The advantage is that you can ensure that you have *no* resource leakage. Basically, you find the parts of the stack that have been overwritten when you gained control, restore them to the values they had before you gained control, and run `ret`.
- **Return to ancestor.** You can normally employ this method by adding a constant to the stack and calling `ret`. If you examine the call stack at the point where you obtain control, you'll probably find some point in the call tree to which you can `ret` without a problem. This works well, for example, in the SQL-UDP bug (that was used by the SQL Slammer worm). You will probably leak some resources, however.
- **Call ancestor.** In a pinch, you might be able to simply call a procedure high up in the call tree, for example, the main thread procedure. In some applications, this works nicely. The downside is that you're likely to leak a lot of resources (sockets, memory, file handles) that might make the program unstable later.

Make the Exploit Stable

It is a good idea to ask yourself a series of questions once you've got the exploit working, so that you can determine whether you need to keep working to make it more stable. Although for some readers it may be enough to be able to point at a single working example of an exploit, if you're actually going to use it in a production environment, you should be aiming for industrial-strength exploits that will work anywhere and won't change the target host in any undesirable ways. This is a good idea in general, but also helps cut overall development time; if you do a good job the first time, you won't have to keep revising your exploit every time a problem occurs.

Here is a quick list; you might want to add more of your own questions:

- Can you run your exploit against a host more than once?
- If you script your exploit and repeatedly run it against a single host, does it fail at some point? Why?
- Can you run multiple copies of your exploit against a host simultaneously?
- If you have a Windows exploit, does it work across all service packs of the target OS?
- Does it work across other Windows OSes? NT/2000/XP/2003?
- If you have a Linux exploit, does it work across multiple distributions? Without needing to specify offsets/versions?

- If you require users to enter a set of offsets in order for your exploit to work, consider hardcoding a set of common platform offsets in your exploit and allowing the user to select based on a friendly name. Even better, use a technique that makes the exploit more platform independent, such as deriving the addresses of `LoadLibrary` and `GetProcAddress` from the PE header in Windows, or not relying on distribution-specific behaviors in Linux.
- What happens if the target host has a well-configured firewall script? Does your exploit hang the target daemon if IPTables or (on Windows) an IPSec filter ruleset blocks the connection?
- What logs does it leave, and how can you clean them up?

Make It Steal the Connection

If you're exploiting a remote bug (and if not, why not?), it's best to reuse the connection on which you came in, the one for your shell, syscall proxy data stream, and so on. Here are some hints for doing this:

- Breakpoint the common socket calls—`accept`, `recv`, `recvfrom`, `send`, `sendto`—and look at where the socket handle is stored. In your shellcode, parse out the handle and reuse it. This might involve using a specific stack or frame offset or possibly brute forcing by using `getpeername` to find the socket that you're talking to.
- In Windows, you might want to breakpoint `ReadFile` and `WriteFile` as well, because they're sometimes used on socket handles.
- If you don't have exclusive access to the socket, don't give up. Find out how access to the socket is being serialized and do the same thing yourself. For example, in Windows, the target process will probably be using an Event, Semaphore, Mutex, or Critical Section. In the first three cases, the threads in question will probably be calling `WaitForSingleObject(Ex)` or `WaitForMultipleObjects(Ex)`, and in the latter case it *must* be calling `EnterCriticalSection`. In all these cases, once you've established the handle (or critical section) that everyone is waiting on, you can wait for access yourself, and play nicely with the other threads.

Conclusion

This chapter covered the tools, files, and programs commonly used when writing exploits. We also went into some of the better papers available for free on the Internet that complement content in this book.