1

BUG HUNTING

Bug hunting is the process of finding bugs in software or hardware. In this book, however, the term *bug hunting* will be used specifically to describe the process of finding security-critical software bugs. Security-critical bugs, also called software security vulnerabilities, allow an attacker to remotely compromise systems, escalate local privileges, cross privilege boundaries, or otherwise wreak havoc on a system.

About a decade ago, hunting for software security vulnerabilities was mostly done as a hobby or as a way to gain media attention. Bug hunting found its way into the mainstream when people realized that it's possible to profit from vulnerabilities.[1]

Software security vulnerabilities, and programs that take advantage of such vulnerabilities (known as *exploits*), get a lot of press coverage. In addition, numerous books and Internet resources describe the process of exploiting these vulnerabilities, and there are perpetual debates over how to disclose bug findings. Despite all this, surprisingly little has been published on the bug-hunting process itself. Although terms like *software vulnerability* or *exploit* are widely used, many people—even many information security professionals—don't know how bug hunters find security vulnerabilities in software.

If you ask 10 different bug hunters how they search through software for security-related bugs, you will most likely get 10 different

answers. This is one of the reasons why there is not, and probably will never be, a "cookbook" for bug hunting. Rather than trying and failing to write a book of generalized instructions, I will describe the approaches and techniques that I used to find specific bugs in real-life software. Hopefully this book will help you develop your own style so you can find some interesting security-critical software bugs.

1.1 For Fun and Profit

People who hunt for bugs have a variety of goals and motivations. Some independent bug hunters want to improve software security, while others seek personal gain in the form of fame, media attention, payment, or employment. A company might want to find bugs to use them as fodder for marketing campaigns. Of course, there are always the bad apples who want to find new ways to break into systems or networks. On the other hand, some people simply do it for fun—or to save the world. ☺

1.2 Common Techniques

Although no formal documentation exists that describes the standard bug-hunting process, common techniques do exist. These techniques can be split into two categories: *static* and *dynamic*. In static analysis, also referred to as *static code analysis*, the source code of the software, or the disassembly of a binary, is examined but not executed. Dynamic analysis, on the other hand, involves debugging or fuzzing the target software while it's executing. Both techniques have pros and cons, and most bug hunters use a combination of static and dynamic techniques.

My Preferred Techniques

Most of the time, I prefer the static analysis approach. I usually read the source code or disassembly of the target software line by line and try to understand it. However, reading all the code from beginning to end is generally not practical. When I'm looking for bugs, I typically start by trying to identify where user-influenced input data enters the software through an interface to the outside world. This could be network data, file data, or data from the execution environment, to name just a few examples.

Next, I study the different ways that the input data can travel through the software, while looking for any potentially exploitable code that acts on the data. Sometimes I'm able to identify these entry points solely by reading the source code (see Chapter 2) or the disassembly (see Chapter 6). In other cases, I have to combine static analysis with the results of debugging the target software (see Chapter 5) to find the input-handling code. I also tend to combine static and dynamic approaches when developing an exploit.

After I've found a bug, I want to prove if it's actually exploitable, so I attempt to build an exploit for it. When I build such an exploit, I spend most of my time in the debugger.

Potentially Vulnerable Code Locations

This is only one approach to bug hunting. Another tactic for finding potentially vulnerable locations in the code is to look at the code near "unsafe" C/C++ library functions, such as `strcpy()` and `strcat()`, in search of possible buffer overflows. Alternatively, you could search the disassembly for `movsx` assembler instructions in order to find sign-extension vulnerabilities. If you find a potentially vulnerable code location, you can then trace backward through the code to see whether these code fragments expose any vulnerabilities accessible from an application entry point. I rarely use this approach, but other bug hunters swear by it.

Fuzzing

A completely different approach to bug hunting is known as *fuzzing*. Fuzzing is a dynamic-analysis technique that consists of testing an application by providing it with malformed or unexpected input. Though I'm not an expert in fuzzing and fuzzing frameworks—I know bug hunters who have developed their own fuzzing frameworks and find most of their bugs with their fuzzing tools—I do use this approach from time to time to determine where user-influenced input enters the software and sometimes to find bugs (see Chapter 8).

You may be wondering how fuzzing can be used to identify where user-influenced input enters the software. Imagine you have a complex application in the form of a binary that you want to examine for bugs. It isn't easy to identify the entry points of such complex applications, but complex software often tends to crash while processing malformed input data. This can hold true for software that parses data files, such as office products, media players, or web browsers. Most of these crashes are not security relevant (e.g., a division-by-zero bug in a browser), but they often provide an entry point where I can start looking for user-influenced input data.

Further Reading

These are only a few of the available techniques and approaches that can be used to find bugs in software. For more information on finding security vulnerabilities in source code, I recommend Mark Dowd, John McDonald, and Justin Schuh's *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities* (Addison-Wesley, 2007). If you want more information about fuzzing, see Michael Sutton, Adam Greene, and Pedram Amini's *Fuzzing: Brute Force Vulnerability Discovery* (Addison-Wesley, 2007).

1.3 Memory Errors

The vulnerabilities described in this book have one thing in common: They all lead to exploitable memory errors. Such memory errors occur when a process, a thread, or the kernel is

- Using memory it does not own (e.g., NULL pointer dereferences, as described in Section A.2)
- Using more memory than has been allocated (e.g., buffer overflows, as described in Section A.1)
- Using uninitialized memory (e.g., uninitialized variables)[2]
- Using faulty heap-memory management (e.g., double frees)[3]

Memory errors typically happen when powerful C/C++ features like explicit memory management or pointer arithmetic are used incorrectly.

A subcategory of memory errors, called *memory corruption*, happens when a process, a thread, or the kernel modifies a memory location that it doesn't own or when the modification corrupts the state of the memory location.

If you're not familiar with such memory errors, I suggest you have a look at Sections A.1, A.2, and A.3. These sections describe the basics of the programming errors and vulnerabilities discussed in this book.

In addition to exploitable memory errors, dozens of other vulnerability classes exist. These include logical errors and web-specific vulnerabilities like cross-site scripting, cross-site request forgery, and SQL injection, to name just a few. However, these other vulnerability classes are not the subject of this book. All the bugs discussed in this book were the result of exploitable memory errors.

1.4 Tools of the Trade

When searching for bugs, or building exploits to test them, I need a way to see inside the workings of applications. I most often use debuggers and disassemblers to gain that inside view.

Debuggers

A debugger normally provides methods to attach to user space processes or the kernel, write and read values to and from registers and memory, and to control program flow using features such as breakpoints or single-stepping. Each operating system typically ships with its own debugger, but several third-party debuggers are available as well. Table 1-1 lists the different operating system platforms and the debuggers used in this book.

Table 1-1: Debuggers Used in This Book

Operating system	Debugger	Kernel debugging
Microsoft Windows	WinDbg (the official debugger from Microsoft)	yes
	OllyDbg and its variant Immunity Debugger	no
Linux	The GNU Debugger (gdb)	yes
Solaris	The Modular Debugger (mdb)	yes
Mac OS X	The GNU Debugger (gdb)	yes
Apple iOS	The GNU Debugger (gdb)	yes

These debuggers will be used to identify, analyze and exploit the vulnerabilities that I discovered. See also Sections B.1, B.2, and B.4 for some debugger command cheat sheets.

Disassemblers

If you want to audit an application and don't have access to the source code, you can analyze the program binaries by reading the application's assembly code. Although debuggers have the ability to disassemble the code of a process or the kernel, they usually are not especially easy or intuitive to work with. A program that fills this gap is the Interactive Disassembler Professional, better known as IDA Pro.[4] IDA Pro supports more than 50 families of processors and provides full interactivity, extensibility, and code graphing. If you want to audit a program binary, IDA Pro is a must-have. For an exhaustive treatment of IDA Pro and all of its features, consult Chris Eagle's *The IDA Pro Book,* 2nd edition (No Starch Press, 2011).

1.5 EIP = 41414141

← Instruction pointer/ Program counter:
• EIP—32-bit instruction pointer (IA-32)
• RIP—64-bit instruction pointer (Intel 64)
• R15 or PC—ARM architecture as used on Apple's iPhone

To illustrate the security implications of the bugs that I found, I will discuss the steps needed to gain control of the execution flow of the vulnerable program by controlling the instruction pointer (IP) of the CPU. The instruction pointer or program counter (PC) register contains the offset in the current code segment for the next instruction to be executed.[5] If you gain control of this register, you fully control the execution flow of the vulnerable process. To demonstrate instruction pointer control, I will modify the register to values like `0x41414141` (hexadecimal representation of ASCII "`AAAA`"), `0x41424344` (hexadecimal representation of ASCII "`ABCD`"), or something similar. So if you see `EIP = 41414141`