Part

Introduction to Exploitation: Linux on x86

Welcome to the Part I of the *Shellcoder's Handbook Second Edition: Discovering and Exploiting Security Holes*. This part is an introduction to vulnerability discovery and exploitation. It is organized in a manner that will allow you to learn exploitation on various fictitious sample code structures created specifically for this book to aid in the learning process, as well as real-life, in-the-wild, vulnerabilities.

You will learn the details of exploitation under Linux running on an Intel 32-bit (IA32 or x86) processor. The discovery and exploitation of vulnerabilities on Linux/IA32 is the easiest and most straightforward to comprehend. This is why we have chosen to start with Linux/IA32. Linux is easiest to understand from a hacker's point of view because you have solid, reliable, internal operating system structures to work with when exploiting.

After you have a solid understanding of these concepts and have worked through the example code, you are graduated to increasingly difficult vulnerability discovery and exploitation scenarios in subsequent Parts. We work through stack buffer overflows in Chapter 2, introductory shellcoding in Chapter 3, format string overflows in Chapter 4, and finally finish up the part with heap-based buffer overflow hacking techniques for the Linux platform in Chapter 5. Upon completion of this part, you will be well on your way to understanding vulnerability development and exploitation.

CHAPTER 1

Before You Begin

This chapter goes over the concepts you need to understand in order to make sense of the rest of this book. Much like some of the reading required for a college course, the material covered here is introductory and hopefully already known to you. This chapter is by no means an attempt to cover everything you need to know; rather, it should serve as jumping off point to the other chapters.

You should read through this chapter as a refresher. If you find concepts that are foreign to you, we suggest that you mark these down as areas on which you need to do more research. Take the time to learn about these concepts before venturing to later chapters.

You will find many of the sample code and code fragments in this book on *The Shellcoder's Handbook* Web site (`http://www.wiley.com/go/shellcodershandbook`); you can copy and paste these samples into your favorite text editor to save time when working on examples.

Basic Concepts

To understand the content of this book, you need a well-developed understanding of computer languages, operating systems, and architectures. If you do not understand how something works, it is difficult to detect that it is malfunctioning. This holds true for computers as well as for discovering and exploiting security holes.

Before you begin to understand the concepts, you must be able to speak the language. You will need to know a few definitions, or terms, that are part of the vernacular of security researchers so that you can better apply the concepts in this book:

Vulnerability (n.): A flaw in a system's security that can lead to an attacker utilizing the system in a manner other than the designer intended. This can include impacting the availability of the system, elevating access privileges to an unintended level, complete control of the system by an unauthorized party, and many other possibilities. Also known as a *security hole* or *security bug*.

Exploit (v.): To take advantage of a vulnerability so that the target system reacts in a manner other than which the designer intended.

Exploit (n.): The tool, set of instructions, or code that is used to take advantage of a vulnerability. Also known as a *Proof of Concept* (POC).

0day (n.): An exploit for a vulnerability that has not been publicly disclosed. Sometimes used to refer to the vulnerability itself.

Fuzzer (n.): A tool or application that attempts all, or a wide range of, unexpected input values to a system. The purpose of a fuzzer is to determine whether a bug exists in the system, which could later be exploited without having to fully know the target system's internal functioning.

Memory Management

To use this book, you will need to understand modern memory management, specifically for the Intel Architecture, 32 Bit (IA32). Linux on IA32 is covered exclusively in the first section of this book and used in the introductory chapters. You will need to understand how memory is managed, because most security holes described in this book come from *overwriting* or *overflowing* one portion of memory into another.

INSTRUCTIONS AND DATA

A modern computer makes no real distinction between instructions and data. If a processor can be fed instructions when it should be seeing data, it will happily go about executing the passed instructions. This characteristic makes system exploitation possible. This book teaches you how to insert instructions when the system designer expected data. You will also use the concept of overflowing to overwrite the designer's instructions with your own. The goal is to gain control of execution.

When a program is executed, it is laid out in an organized manner—various elements of the program are mapped into memory. First, the operating system creates an address space in which the program will run. This address space includes the actual program instructions as well as any required data.

Next, information is loaded from the program's executable file to the newly created address space. There are three types of segments: `.text`, `.bss`, and `.data`. The `.text` segment is mapped as read-only, whereas `.data` and `.bss` are writable. The `.bss` and `.data` segments are reserved for global variables. The `.data` segment contains static initialized data, and the `.bss` segment contains uninitialized data. The final segment, `.text`, holds the program instructions.

Finally, the *stack* and the *heap* are initialized. The stack is a data structure, more specifically a *Last In First Out* (LIFO) data structure, which means that the most recent data placed, or pushed, onto the stack is the next item to be removed, or popped, from the stack. A LIFO data structure is ideal for storing transitory information, or information that does not need to be stored for a lengthy period of time. The stack stores local variables, information relating to function calls, and other information used to clean up the stack after a function or procedure is called.

Another important feature of the stack is that it *grows down* the address space: as more data is added to the stack, it is added at increasingly lower address values.

The heap is another data structure used to hold program information, more specifically, dynamic variables. The heap is (roughly) a *First In First Out* (FIFO) data structure. Data is placed and removed from the heap as it builds. The heap *grows up* the address space: As data is added to the heap, it is added at an increasingly higher address value, as shown in the following memory space diagram.

↑ Lower addresses (0x08000000)
`Shared libraries`
`.text`
`.bss`
`Heap (grows ↓)`
`Stack (grows ↑)`
`env pointer`
`Argc`
↓ Higher addresses (0xbfffffff)

Memory management presented in this section must be understood on a much deeper, more detailed level to fully comprehend, and more importantly, apply what is contained in this book. Check the first half of Chapter 15 for places to learn more about memory management. You can also pay a visit to `http://linux-mm.org/` for more detailed information on memory management on Linux. Understanding memory management concepts will help you

better comprehend the programming language you will use to manipulate them—assembly.

Assembly

Knowledge of assembly language specific to IA32 is required in order to understand much of this book. Much of the bug discovery process involves interpreting and understanding assembly, and much of this book focuses on assembly with the 32-bit Intel processor. Exploiting security holes requires a firm grasp of assembly language, because most exploits will require you to write (or modify existing) code in assembly.

Because systems other than IA32 are important, but can be somewhat more difficult to exploit, this book also covers bug discovery and exploitation on other processor families. If you are planning to pursue security research on other platforms, it is important for you to have a strong understanding of assembly specific to your chosen architecture.

If you are not well versed in or have no experience with assembly, you will first need to learn number systems (specifically hexadecimal), data sizes, and number sign representations. These computer-engineering concepts can be found in most college-level computer architecture books.

Registers

Understanding how the registers work on an IA32 processor and how they are manipulated via assembly is essential for vulnerability development and exploitation. Registers can be accessed, read, and changed with assembly.

Registers are memory, usually connected directly to circuitry for performance reasons. They are responsible for manipulations that allow modern computers to function, and can be manipulated with assembly instructions. From a high level, registers can be grouped into four categories:

- General purpose
- Segment
- Control
- Other

General-purpose registers are used to perform a range of common mathematical operations. They include registers such as `EAX`, `EBX`, and `ECX` for the IA32, and can be used to store data and addresses, offset addresses, perform counting functions, and many other things.

A general-purpose register to take note of is the *extended stack pointer* register (`ESP`) or simply the *stack pointer*. `ESP` points to the memory address where the next stack operation will take place. In order to understand stack overflows in

the next chapter, you should thoroughly understand how ESP is used with common assembly instructions and the effect it has on data stored on the stack.

The next class of register of interest is the *segment* register. Unlike the other registers on an IA32 processor, the segment registers are 16 bit (other registers are 32 bits in size). Segment registers, such as CS, DS, and SS, are used to keep track of segments and to allow backward compatibility with 16-bit applications.

Control registers are used to control the function of the processor. The most important of these registers for the IA32 is the *Extended Instruction Pointer* (EIP) or simply the *Instruction Pointer*. EIP contains the address of the next machine instruction to be executed. Naturally, if you want to control the execution path of a program, which is incidentally what this book is all about, it is important to have the ability to access and change the value stored in the EIP register.

The registers in the *other* category are simply extraneous registers that do not fit neatly into the first three categories. One of these registers is the *Extended Flags* (EFLAGS) register, which comprises many single-bit registers that are used to store the results of various tests performed by the processor.

Once you have a solid understanding of the registers, you can move onto assembly programming itself.

Recognizing C and C++ Code Constructs in Assembly

The C family of programming languages (C, C++, C#) is one of the most widely used, if not the most widely used, genre of programming languages. C is definitely the most popular language for Windows and Unix server applications, which are good targets for vulnerability development. For these reasons, a solid understanding of C is critical.

Along with a broad comprehension of C, you should be able to understand how compiled C code translates into assembly. Understanding how C variables, pointers, functions, and memory allocation are represented by assembly will make the contents of this book much easier to understand.

Let's take some common C and C++ code constructs and see what they look like in assembly. If you have a firm grasp of these examples, you should be ready to move forward with the rest of the book.

Let's look at declaring an integer in C++, then using that same integer for counting:

```
int number;
. . . more code . . .
number++;
```

This could be translated to, in assembly:

```
number dw 0
 . . .more code . . .
mov eax,number
inc eax
mov number,eax
```

We use the Define Word (DW) instruction to define a value for our integer, number. Next we put the value into the EAX register, increment the value in the EAX register by one, and then move this value back into the number integer.

Look at a simple if statement in C++:

```
    int number;
if (number<0)
{
 . . .more code . . .
}
```

Now, look at the same if statement in assembly:

```
number dw 0
mov eax,number
or eax,eax
jge label
<no>
label :<yes>
```

What we are doing here is defining a value for number again with the DW instruction. Then we move the value stored in number into EAX, then we jump to label if number is greater than or equal to zero with Jump if Greater than or Equal to (JGE).

Here's another example, using an array:

```
int array[4];
 . . .more code . . .
array[2]=9;
```

Here we have declared an array, array, and set an array element equal to 9. In assembly we have:

```
array dw 0,0,0,0
 . . .more code . . .
mov ebx,2
mov array[ebx],9
```

In this example, we declare an array, then use the EBX register to move values into the array.

Last, let's take a look at a more complicated example. The code shows how a simple C function looks in assembly. If you can easily understand this example, you are probably ready to move forward to the next chapter.

```
int triangle (int width, in height){

int array[5] = {0,1,2,3,4};
int area;
area = width * height/2;
return (area);

}
```

Here is the same function, but in disassembled form. The following is output from the `gdb` debugger. `gdb` is the GNU project debugger; you can read more about it at `http://www.gnu.org/software/gdb/documentation/`. See if you can match the assembler to the C code:

```
0x8048430 <triangle>:      push   %ebp
0x8048431 <triangle+1>:    mov    %esp, %ebp
0x8048433 <triangle+3>:    push   %edi
0x8048434 <triangle+4>:    push   %esi
0x8048435 <triangle+5>:    sub    $0x30,%esp
0x8048438 <triangle+8>:    lea    0xffffffd8(%ebp), %edi
0x804843b <triangle+11>:    mov   $0x8049508,%esi
0x8048440 <triangle+16>:    cld
0x8048441 <triangle+17>:    mov   $0x30,%esp
0x8048446 <triangle+22>:    repz movsl    %ds:( %esi), %es:( %edi)
0x8048448 <triangle+24>:    mov   0x8(%ebp),%eax
0x804844b <triangle+27>:    mov   %eax,%edx
0x804844d <triangle+29>:    imul  0xc(%ebp),%edx
0x8048451 <triangle+33>:    mov   %edx,%eax
0x8048453 <triangle+35>:    sar   $0x1f,%eax
0x8048456 <triangle+38>:    shr   $0x1f,%eax
0x8048459 <triangle+41>:    lea   (%eax, %edx, 1), %eax
0x804845c <triangle+44>:    sar   %eax
0x804845e <triangle+46>:    mov   %eax,0xffffffd4(%ebp)
0x8048461 <triangle+49>:    mov   0xffffffd4(%ebp),%eax
0x8048464 <triangle+52>:    mov   %eax,%eax
0x8048466 <triangle+54>:    add   $0x30,%esp
0x8048469 <triangle+57>:    pop   %esi
0x804846a <triangle+58>:    pop   %edi
0x804846b <triangle+59>     pop   %ebp
0x804846c <triangle+60>:    ret
```

The main thing the function does is multiply two numbers, so note the `imul` instruction in the middle. Also note the first few instructions—saving `EBP`, and subtracting from `ESP`. The subtraction makes room on the stack for the func-

tion's local variables. It's also worth noting that the function returns its result in the EAX register.

Conclusion

This chapter introduced some basic concepts you need to know in order to understand the rest of this book. You should spend some time reviewing the concepts outlined in this chapter. If you find that you do not have sufficient exposure to assembly language and C or C++, you may need to do some background preparation in order to get full value from the following chapters.