

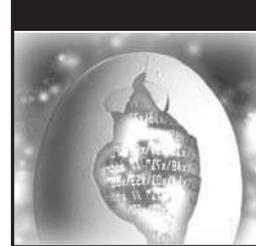
The Shellcoder's Handbook

SECOND EDITION

Discovering and Exploiting Security Holes



Chris Anley\John Heasman\Felix "FX" Linder\Gerardo Richarte



The Shellcoder's Handbook

Discovering and Exploiting Security Holes
Second Edition

Chris Anley
John Heasman
Felix "FX" Linder
Gerardo Richarte

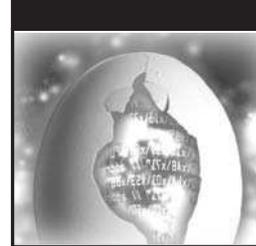
The Shellcoder's Handbook: Discovering and Exploiting Security Holes
(1st Edition) was written by Jack Koziol, David Litchfield, Dave Aitel,
Chris Anley, Sinan Eren, Neel Mehta, and Riley Hassell.



Wiley Publishing, Inc.

The Shellcoder's Handbook

Second Edition



The Shellcoder's Handbook

Discovering and Exploiting Security Holes
Second Edition

Chris Anley
John Heasman
Felix "FX" Linder
Gerardo Richarte

The Shellcoder's Handbook: Discovering and Exploiting Security Holes
(1st Edition) was written by Jack Koziol, David Litchfield, Dave Aitel,
Chris Anley, Sinan Eren, Neel Mehta, and Riley Hassell.



Wiley Publishing, Inc.

The Shellcoder's Handbook, Second Edition: Discovering and Exploiting Security Holes

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2007 by Chris Anley, John Heasman, Felix "FX" Linder, and Gerardo Richarte
Published by Wiley Publishing, Inc., Indianapolis, Indiana
Published simultaneously in Canada

ISBN: 978-0-470-08023-8

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Legal Department, Wiley Publishing, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, (317) 572-3447, fax (317) 572-4355, or online at <http://www.wiley.com/go/permissions>.

Limit of Liability/Disclaimer of Warranty: The publisher and the author make no representations or warranties with respect to the accuracy or completeness of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Website is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or Website may provide or recommendations it may make. Further, readers should be aware that Internet Websites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services or to obtain technical support, please contact our Customer Care Department within the U.S. at (800) 762-2974, outside the U.S. at (317) 572-3993 or fax (317) 572-4002.

Library of Congress Cataloging-in-Publication Data

The shellcoder's handbook : discovering and exploiting security holes / Chris Anley ... [et al.]. — 2nd ed.

p. cm.

ISBN 978-0-470-08023-8 (paper/website)

1. Computer security. 2. Data protection. 3. Risk assessment. I. Anley, Chris.

QA76.9.A25S464 2007

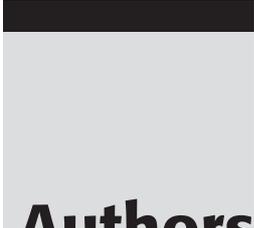
005.8 — dc22

2007021079

Trademarks: Wiley and the Wiley logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

*This book is dedicated to anyone and everyone who understands that
hacking and learning is a way to live your life, not a day job or
semi-ordered list of instructions found in a thick book.*



About the Authors

Chris Anley is a founder and director of NGSSoftware, a security software, consultancy, and research company based in London, England. He is actively involved in vulnerability research and has discovered security flaws in a wide variety of platforms including Microsoft Windows, Oracle, SQL Server, IBM DB2, Sybase ASE, MySQL, and PGP.

John Heasman is the Director of Research at NGSSoftware. He is a prolific security researcher and has published many security advisories in enterprise level software. He has a particular interest in rootkits and has authored papers on malware persistence via device firmware and the BIOS. He is also a co-author of *The Database Hacker's Handbook: Defending Database Servers* (Wiley 2005).

Felix "FX" Linder leads SABRE Labs GmbH, a Berlin-based professional consulting company specializing in security analysis, system design creation, and verification work. Felix looks back at 18 years of programming and over a decade of computer security consulting for enterprise, carrier, and software vendor clients. This experience allows him to rapidly dive into complex systems and evaluate them from a security and robustness point of view, even in atypical scenarios and on arcane platforms. In his spare time, FX works with his friends from the Phenoelit hacking group on different topics, which have included Cisco IOS, SAP, HP printers, and RIM BlackBerry in the past.

Gerardo Richarte has been doing reverse engineering and exploit development for more than 15 years non-stop. In the past 10 years he helped build the technical arm of Core Security Technologies, where he works today. His current duties include developing exploits for Core IMPACT, researching new exploitation techniques and other low-level subjects, helping other exploit writers when things get hairy, and teaching internal and external classes on assembly and exploit writing. As result of his research and as a humble thank

viii About the Authors

you to the community, he has published some technical papers and open source projects, presented in a few conferences, and released part of his training material. He really enjoys solving tough problems and reverse engineering any piece of code that falls in his reach just for the fun of doing it.



Credits

Executive Editor

Carol Long

Senior Development Editor

Kevin Kent

Production Editor

Eric Charbonneau

Project Coordinator, Cover

Adrienne Martinez

Copy Editor

Kim Cofer

Editorial Manager

Mary Beth Wakefield

Production Manager

Tim Tate

**Vice President and Executive
Group Publisher**

Richard Swadley

**Vice President and Executive
Publisher**

Joseph B. Wikert

Compositor

Craig Johnson,
Happenstance Type-O-Rama

Proofreader

Jen Larsen

Indexer

Johnna VanHoose Dinse

Anniversary Logo Design

Richard Pacifico



Acknowledgments

I would first like to thank all of the people that have made this book possible — the (many) authors, of course: Gerardo Richarte, Felix “FX” Linder, John Heasman, Jack Koziol, David Litchfield, Dave Aitel, Sinan Eren, Neel Mehta, and Riley Hassell. Huge thanks are also due to the team at Wiley — our excellent Executive Editor Carol Long and our equally excellent Development Editor Kevin Kent. On a personal note I’d like to thank the team at NGS for a great many hangovers, technical discussions, hangovers, ideas, and hangovers. Finally, I’d like to thank my wife Victoria for her enduring patience, love, and gorgeousness.

— Chris Anley

I would like to thank my friends and family for their unwavering support.

— John Heasman

I would like to thank my friends from Phenoelit, who are still with me despite the turns and detours life takes and despite the strange ideas I have, technical and otherwise. Special thanks in this context go to Mumpi, who is a very good friend and my invaluable support in all kinds of activities. Additional thanks and kudos go to the SABRE Labs team as well as to Halvar Flake, who is responsible for the existence of this team in the first place. Last but not least, I thank Bine for enduring me on a daily basis.

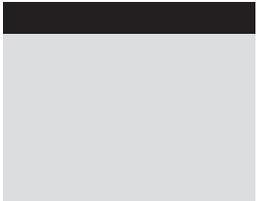
— Felix “FX” Linder

I want to thank those in the community who share what excites them, their ideas and findings, especially the amazing people at Core, past and present, and my pals in the exploit writing team with whom the sudden discovery

xii Acknowledgments

never ends — it is quite often simple and enlightening. I also want to thank Chris and John (co-authors) and Kevin Kent from Wiley Publishing, who all took the time to go through my entangled English, turning it more than just readable. And I want to thank Chinchin, my love, who's always by my side, asking me questions when I need them, listening when I talk or am quiet, and supporting me, always.

— Gerardo Richarte



Contents

About the Authors	vii
Acknowledgments	xi
Introduction to the Second Edition	xxiii
Part I Introduction to Exploitation: Linux on x86	
Chapter 1 Before You Begin	3
Basic Concepts	3
Memory Management	4
Assembly	6
Recognizing C and C++ Code Constructs in Assembly	7
Conclusion	10
Chapter 2 Stack Overflows	11
Buffers	12
The Stack	13
Functions and the Stack	15
Overflowing Buffers on the Stack	18
Controlling EIP	22
An Interesting Diversion	23
Using an Exploit to Get Root Privileges	25
The Address Problem	27
The NOP Method	33
Defeating a Non-Executable Stack	35
Return to libc	35
Conclusion	39

Chapter 3	Shellcode	41
	Understanding System Calls	42
	Writing Shellcode for the exit() Syscall	44
	Injectable Shellcode	48
	Spawning a Shell	50
	Conclusion	59
Chapter 4	Introduction to Format String Bugs	61
	Prerequisites	61
	What Is a Format String?	61
	What Is a Format String Bug?	63
	Format String Exploits	68
	Crashing Services	69
	Information Leakage	70
	Controlling Execution for Exploitation	75
	Why Did This Happen?	84
	Format String Technique Roundup	85
	Conclusion	88
Chapter 5	Introduction to Heap Overflows	89
	What Is a Heap?	90
	How a Heap Works	91
	Finding Heap Overflows	91
	Basic Heap Overflows	93
	Intermediate Heap Overflows	98
	Advanced Heap Overflow Exploitation	105
	Conclusion	107
Part II	Other Platforms—Windows, Solaris, OS/X, and Cisco	
Chapter 6	The Wild World of Windows	111
	How Does Windows Differ from Linux?	111
	Win32 API and PE-COFF	112
	Heaps	114
	Threading	115
	The Genius and Idiocy of the Distributed Common	
	Object Model and DCE-RPC	116
	Recon	118
	Exploitation	120
	Tokens and Impersonation	120
	Exception Handling under Win32	122
	Debugging Windows	124
	Bugs in Win32	124
	Writing Windows Shellcode	125
	A Hacker's Guide to the Win32 API	126
	A Windows Family Tree from the Hacker's Perspective	126
	Conclusion	127

Chapter 7	Windows Shellcode	129
	Syntax and Filters	129
	Setting Up	131
	Parsing the PEB	132
	Heapoverflow.c Analysis	132
	Searching with Windows Exception Handling	148
	Popping a Shell	153
	Why You Should Never Pop a Shell on Windows	153
	Conclusion	154
 Chapter 8	 Windows Overflows	 155
	Stack-Based Buffer Overflows	156
	Frame-Based Exception Handlers	156
	Abusing Frame-Based Exception Handling on Windows 2003 Server	161
	A Final Note about Frame-Based Handler Overwrites	166
	Stack Protection and Windows 2003 Server	166
	Heap-Based Buffer Overflows	173
	The Process Heap	173
	Dynamic Heaps	173
	Working with the Heap	173
	How the Heap Works	174
	Exploiting Heap-Based Overflows	178
	Overwrite Pointer to RtlEnterCriticalSection in the PEB	178
	Overwrite Pointer to Unhandled Exception Filter	185
	Repairing the Heap	191
	Other Aspects of Heap-Based Overflows	193
	Wrapping Up the Heap	194
	Other Overflows	194
	.data Section Overflows	194
	TEB/PEB Overflows	196
	Exploiting Buffer Overflows and Non-Executable Stacks	197
	Conclusion	203
 Chapter 9	 Overcoming Filters	 205
	Writing Exploits for Use with an Alphanumeric Filter	205
	Writing Exploits for Use with a Unicode Filter	209
	What Is Unicode?	210
	Converting from ASCII to Unicode	210
	Exploiting Unicode-Based Vulnerabilities	211
	The Available Instruction Set in Unicode Exploits	212
	The Venetian Method	213
	An ASCII Venetian Implementation	214
	Decoder and Decoding	218
	The Decoder Code	219
	Getting a Fix on the Buffer Address	220
	Conclusion	221

Chapter 10	Introduction to Solaris Exploitation	223
	Introduction to the SPARC Architecture	224
	Registers and Register Windows	224
	The Delay Slot	227
	Synthetic Instructions	228
	Solaris/SPARC Shellcode Basics	228
	Self-Location Determination and SPARC Shellcode	228
	Simple SPARC exec Shellcode	229
	Useful System Calls on Solaris	230
	NOP and Padding Instructions	231
	Solaris/SPARC Stack Frame Introduction	231
	Stack-Based Overflow Methodologies	232
	Arbitrary Size Overflow	232
	Register Windows and Stack Overflow Complications	233
	Other Complicating Factors	233
	Possible Solutions	234
	Off-By-One Stack Overflow Vulnerabilities	234
	Shellcode Locations	235
	Stack Overflow Exploitation In Action	236
	The Vulnerable Program	236
	The Exploit	238
	Heap-Based Overflows on Solaris/SPARC	241
	Solaris System V Heap Introduction	242
	Heap Tree Structure	242
	Basic Exploit Methodology (t_delete)	263
	Standard Heap Overflow Limitations	266
	Targets for Overwrite	267
	Other Heap-Related Vulnerabilities	270
	Off-by-One Overflows	270
	Double Free Vulnerabilities	270
	Arbitrary Free Vulnerabilities	271
	Heap Overflow Example	271
	The Vulnerable Program	272
	Other Solaris Exploitation Techniques	276
	Static Data Overflows	276
	Bypassing the Non-Executable Stack Protection	276
	Conclusion	277
Chapter 11	Advanced Solaris Exploitation	279
	Single Stepping the Dynamic Linker	281
	Various Style Tricks for Solaris SPARC Heap Overflows	296
	Advanced Solaris/SPARC Shellcode	299
	Conclusion	311

Chapter 12	OS X Shellcode	313
	OS X Is Just BSD, Right?	314
	Is OS X Open Source?	314
	OS X for the Unix-aware	315
	Password Cracking	316
	OS X PowerPC Shellcode	316
	OS X Intel Shellcode	324
	Example Shellcode	326
	ret2libc	327
	ret2str(l)cpy	329
	OS X Cross-Platform Shellcode	332
	OS X Heap Exploitation	333
	Bug Hunting on OS X	335
	Some Interesting Bugs	335
	Essential Reading for OS X Exploits	337
	Conclusion	338
Chapter 13	Cisco IOS Exploitation	339
	An Overview of Cisco IOS	339
	Hardware Platforms	340
	Software Packages	340
	IOS System Architecture	343
	Vulnerabilities in Cisco IOS	346
	Protocol Parsing Code	347
	Services on the Router	347
	Security Features	348
	The Command-Line Interface	348
	Reverse Engineering IOS	349
	Taking the Images Apart	349
	Diffing IOS Images	350
	Runtime Analysis	351
	Exploiting Cisco IOS	357
	Stack Overflows	357
	Heap Overflows	359
	Shellcodes	364
	Conclusion	373
Chapter 14	Protection Mechanisms	375
	Protections	375
	Non-Executable Stack	376
	W^X (Either Writable or Executable) Memory	381
	Stack Data Protection	388
	AAAS: ASCII Armored Address Space	394
	ASLR: Address Space Layout Randomization	396
	Heap Protections	399
	Windows SEH Protections	407
	Other Protections	411

Implementation Differences	413
Windows	413
Linux	417
OpenBSD	421
Mac OS X	422
Solaris	423
Conclusion	425
Part III	Vulnerability Discovery
Chapter 15	Establishing a Working Environment 429
What You Need for Reference	430
What You Need for Code	430
gcc	430
gdb	430
NASM	431
WinDbg	431
OllyDbg	431
Visual C++	431
Python	432
What You Need for Investigation	432
Useful Custom Scripts/Tools	432
All Platforms	434
Unix	434
Windows	435
What You Need to Know	436
Paper Archives	438
Optimizing Shellcode Development	439
Plan the Exploit	439
Write the Shellcode in Inline Assembler	439
Maintain a Shellcode Library	441
Make It Continue Nicely	441
Make the Exploit Stable	442
Make It Steal the Connection	443
Conclusion	443
Chapter 16	Fault Injection 445
Design Overview	447
Input Generation	447
Fault Injection	450
Modification Engines	450
Fault Delivery	455
Nagel Algorithm	455
Timing	455
Heuristics	456
Stateless versus State-Based Protocols	456
Fault Monitoring	456
Using a Debugger	457
FaultMon	457

Putting It Together	458
Conclusion	459
Chapter 17 The Art of Fuzzing	461
General Theory of Fuzzing	461
Static Analysis versus Fuzzing	466
Fuzzing Is Scalable	466
Weaknesses in Fuzzers	468
Modeling Arbitrary Network Protocols	469
Other Fuzzer Possibilities	469
Bit Flipping	469
Modifying Open Source Programs	470
Fuzzing with Dynamic Analysis	470
SPIKE	471
What Is a Spike?	471
Why Use the SPIKE Data Structure to Model Network Protocols?	472
Other Fuzzers	480
Conclusion	480
Chapter 18 Source Code Auditing: Finding Vulnerabilities in C-Based Languages	481
Tools	482
Cscope	482
Ctags	483
Editors	483
Cbrowser	484
Automated Source Code Analysis Tools	484
Methodology	485
Top-Down (Specific) Approach	485
Bottom-Up Approach	485
Selective Approach	485
Vulnerability Classes	486
Generic Logic Errors	486
(Almost) Extinct Bug Classes	487
Format Strings	487
Generic Incorrect Bounds-Checking	489
Loop Constructs	490
Off-by-One Vulnerabilities	490
Non-Null Termination Issues	492
Skipping Null-Termination Issues	493
Signed Comparison Vulnerabilities	494
Integer-Related Vulnerabilities	495
Different-Sized Integer Conversions	497
Double Free Vulnerabilities	498
Out-of-Scope Memory Usage Vulnerabilities	499
Uninitialized Variable Usage	499
Use After Free Vulnerabilities	500
Multithreaded Issues and Re-Entrant Safe Code	500

	Beyond Recognition: A Real Vulnerability versus a Bug	501
	Conclusion	501
Chapter 19	Instrumented Investigation: A Manual Approach	503
	Philosophy	503
	Oracle extproc Overflow	504
	Common Architectural Failures	508
	Problems Happen at Boundaries	508
	Problems Happen When Data Is Translated	509
	Problems Cluster in Areas of Asymmetry	511
	Problems Occur When Authentication and Authorization Are Confused	512
	Problems Occur in the Dumbest Places	512
	Bypassing Input Validation and Attack Detection	513
	Stripping Bad Data	513
	Using Alternate Encodings	514
	Using File-Handling Features	515
	Evading Attack Signatures	517
	Defeating Length Limitations	517
	Windows 2000 SNMP DOS	520
	Finding DOS Attacks	521
	SQL-UDP	522
	Conclusion	523
Chapter 20	Tracing for Vulnerabilities	525
	Overview	526
	A Vulnerable Program	527
	Component Design	529
	Building VulnTrace	538
	Using VulnTrace	543
	Advanced Techniques	546
	Conclusion	548
Chapter 21	Binary Auditing: Hacking Closed Source Software	549
	Binary versus Source-Code Auditing: The Obvious Differences	550
	IDA Pro—The Tool of the Trade	550
	Features: A Quick Crash Course	551
	Debugging Symbols	552
	Binary Auditing Introduction	552
	Stack Frames	552
	Calling Conventions	554
	Compiler-Generated Code	556
	memcpy-Like Code Constructs	560
	strlen-Like Code Constructs	560
	C++ Code Constructs	561
	The this Pointer	561
	Reconstructing Class Definitions	562
	vtables	562
	Quick but Useful Tidbits	563

Manual Binary Analysis	563
Quick Examination of Library Calls	564
Suspicious Loops and Write Instructions	564
Higher-Level Understanding and Logic Bugs	565
Graphical Analysis of Binaries	566
Manual Decompilation	566
Binary Vulnerability Examples	566
Microsoft SQL Server Bugs	566
LSD's RPC-DCOM Vulnerability	567
IIS WebDAV Vulnerability	568
Conclusion	570
Part IV Advanced Materials	
Chapter 22 Alternative Payload Strategies	573
Modifying the Program	574
The SQL Server 3-Byte Patch	575
The MySQL 1-Bit Patch	578
OpenSSH RSA Authentication Patch	580
Other Runtime Patching Ideas	581
GPG 1.2.2 Randomness Patch	583
Upload and Run (or Proglet Server)	584
Syscall Proxies	584
Problems with Syscall Proxies	587
Conclusion	596
Chapter 23 Writing Exploits that Work in the Wild	597
Factors in Unreliability	597
Magic Numbers	597
Versioning	598
Shellcode Problems	599
Countermeasures	601
Preparation	602
Brute Forcing	602
Local Exploits	603
OS/Application Fingerprinting	603
Information Leaks	605
Conclusion	606
Chapter 24 Attacking Database Software	607
Network Layer Attacks	608
Application Layer Attacks	618
Running Operating System Commands	619
Microsoft SQL Server	619
Oracle	620
IBM DB2	621
Exploiting Overruns at the SQL Level	623
SQL Functions	623
Conclusion	625

Chapter 25	Unix Kernel Overflows	627
	Kernel Vulnerability Types	627
	0day Kernel Vulnerabilities	636
	OpenBSD exec_ibcs2_coff_prep_zmagic() Stack Overflow	636
	The Vulnerability	638
	Solaris vfs_getvfssw() Loadable Kernel Module	
	Traversal Vulnerability	642
	The sysfs() System Call	644
	The mount() System Call	645
	Conclusion	646
Chapter 26	Exploiting Unix Kernel Vulnerabilities	647
	The exec_ibcs2_coff_prep_zmagic() Vulnerability	647
	Calculating Offsets and Breakpoints	652
	Overwriting the Return Address and Redirecting Execution	654
	Locating the Process Descriptor (or the Proc Structure)	655
	Kernel Mode Payload Creation	658
	Returning Back from Kernel Payload	659
	Getting root (uid=0)	665
	Solaris vfs_getvfssw() Loadable Kernel	
	Module Path Traversal Exploit	672
	Crafting the Exploit	673
	The Kernel Module to Load	674
	Getting root (uid=0)	678
	Conclusion	678
Chapter 27	Hacking the Windows Kernel	681
	Windows Kernel Mode Flaws—An Increasingly Hunted Species	681
	Introduction to the Windows Kernel	682
	Common Kernel-Mode Programming Flaws	683
	Stack Overflows	684
	Heap Overflows	688
	Insufficient Validation of User-Mode Addresses	688
	Repurposing Attacks	689
	Shared Object Attacks	689
	Windows System Calls	690
	Understanding System Calls	690
	Attacking System Calls	692
	Communicating with Device Drivers	693
	I/O Control Code Components	693
	Finding Flaws in IOCTL Handlers	694
	Kernel-Mode Payloads	695
	Elevating a User-Mode Process	696
	Running an Arbitrary User-Mode Payload	699
	Subverting Kernel Security	701
	Installing a Rootkit	703
	Essential Reading for Kernel Shellcoders	703
	Conclusion	704
	Index	705



Introduction to the Second Edition

Wherever terms have a shifting meaning, independent sets of considerations are liable to become complicated together, and reasonings and results are frequently falsified.

— *Ada Augusta, Countess of Lovelace,*
from her notes on “Sketch of The Analytical Engine,” 1842

You have in your hands *The Shellcoder’s Handbook Second Edition: Discovering and Exploiting Security Holes*. The first edition of this volume attempted to show the reader how security vulnerabilities are discovered and exploited, and this edition holds fast to that same objective. If you’re a skilled network auditor, software developer, or sysadmin and you want to understand how bugs are found and how exploits work at the lowest level, you’ve come to the right place.

So what’s this book about? Well, the preceding quotation more or less sums it up. This book is mostly concerned with arbitrary code execution vulnerabilities, by which we mean bugs, that allow attackers to run code of their choice on the target machine. This generally happens when a program interprets a piece of data as a part of the program — part of an http “Host” header becomes a return address, part of an email address becomes a function pointer, and so on. The program ends up executing the data the attacker supplied with disastrous effects. The architecture of modern processors, operating systems, and compilers lends itself toward this kind of problem — as the good Countess wrote, “the symbols of operation are frequently also the symbols of the results of operations.” Of course, she was writing about the difficulty of discussing mathematics when the number “5” might also mean “raised to the power of 5” or “the fifth element of a series,” but the basic idea is the same. If you confuse code and data, you’re in a world of trouble. So, this book is about code and data, and what happens when the two become confused.

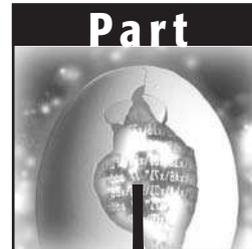
This subject area has become much more complicated since the first edition of this volume was published; the world has moved on since 2004. It's now commonplace for compilers and operating systems to have built-in measures that protect against the types of vulnerabilities this book is mostly concerned with, though it's also true to say that these measures are far from perfect. Nor does the supply of arbitrary-code execution bugs look to be drying up any time soon, despite advances in methods for finding them — if you check out the U.S. National Vulnerability Database Web site (nvd.nist.gov), click “statistics” and select “buffer overflow,” you'll see that buffer overflows continue to increase in number, running at around 600 per year in 2005 and 2006, with 2007 on course to match or exceed that.

So it's clear that we still need to know about these bugs and how they're exploited — in fact, there's a strong argument that it's more important to know about the precise mechanisms now that we have so many partial defenses to choose from when considering how to protect ourselves. If you're auditing a network, a working exploit will give you 100 percent confidence in your assessment, and if you're a software developer, creating proof-of-concept exploits can help understand which bugs need to be fixed first. If you're purchasing a security product, knowing how to get around a non-executable stack, exploit a tricky heap overflow, or write your own exploit encoder will help you to make a better judgment of the quality of the various vendors. In general, knowledge is preferable to ignorance. The bad guys already know this stuff; the network-auditing, software-writing, network-managing public should know it, too.

So why is this book different? Well, first, the authors find and exploit bugs as part of their day jobs. We're not just writing about this stuff; we're doing it on a daily basis. Second, you'll not see us writing too much about tools. Most of the content of this book is concerned with the raw meat of security bugs — assembler, source code, the stack, the heap, and so on. These ideas allow you to write tools rather than just use tools written by others. Finally, there's a question of focus and attitude. It isn't written down in any particular paragraph, but the message that shines out through the whole of this book is that you should experiment, explore, and try to understand the systems you're running. You'll find a lot of interesting stuff that way.

So, without further ado, here's the second edition of *The Shellcoder's Handbook*. I hope you enjoy it, I hope it's useful, and I hope you use it to do some good. If you have any comments, criticisms, or suggestions, please let me know.

Cheers,
Chris Anley



Introduction to Exploitation: Linux on x86

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

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

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

Before You Begin

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

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

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

Basic Concepts

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

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

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

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

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

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

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

Memory Management

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

INSTRUCTIONS AND DATA

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

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

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

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

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

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

```

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

```

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

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

Assembly

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

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

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

Registers

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

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

- General purpose
- Segment
- Control
- Other

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

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

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

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

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

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

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

Recognizing C and C++ Code Constructs in Assembly

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

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

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

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

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

This could be translated to, in assembly:

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

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

Look at a simple *if* statement in C++:

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

Now, look at the same *if* statement in assembly:

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

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

Here's another example, using an array:

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

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

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

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

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

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

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

}
```

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

```
0x8048430 <triangle>:      push   %ebp
0x8048431 <triangle+1>:      mov    %esp, %ebp
0x8048433 <triangle+3>:      push   %edi
0x8048434 <triangle+4>:      push   %esi
0x8048435 <triangle+5>:      sub    $0x30, %esp
0x8048438 <triangle+8>:      lea   0xfffffd8(%ebp), %edi
0x804843b <triangle+11>:     mov    $0x8049508, %esi
0x8048440 <triangle+16>:     cld
0x8048441 <triangle+17>:     mov    $0x30, %esp
0x8048446 <triangle+22>:     repz  movsl  %ds:( %esi), %es:( %edi)
0x8048448 <triangle+24>:     mov    0x8(%ebp), %eax
0x804844b <triangle+27>:     mov    %eax, %edx
0x804844d <triangle+29>:     imul  0xc(%ebp), %edx
0x8048451 <triangle+33>:     mov    %edx, %eax
0x8048453 <triangle+35>:     sar   $0x1f, %eax
0x8048456 <triangle+38>:     shr   $0x1f, %eax
0x8048459 <triangle+41>:     lea   (%eax, %edx, 1), %eax
0x804845c <triangle+44>:     sar   %eax
0x804845e <triangle+46>:     mov    %eax, 0xfffffd4(%ebp)
0x8048461 <triangle+49>:     mov    0xfffffd4(%ebp), %eax
0x8048464 <triangle+52>:     mov    %eax, %eax
0x8048466 <triangle+54>:     add   $0x30, %esp
0x8048469 <triangle+57>:     pop   %esi
0x804846a <triangle+58>:     pop   %edi
0x804846b <triangle+59>:     pop   %ebp
0x804846c <triangle+60>:     ret
```

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

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

Conclusion

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

Stack Overflows

Stack-based buffer overflows have historically been one of the most popular and best understood methods of exploiting software. Tens, if not hundreds, of papers have been written on stack overflow techniques on all manner of popular architectures. One of the most frequently referred to, and likely the first public discourse on stack overflows, is Aleph One's "Smashing the Stack for Fun and Profit." Written in 1996 and published in *Phrack* magazine, the paper explained for the first time in a clear and concise manner how buffer overflow vulnerabilities are possible and how they can be exploited. We recommend that you read the paper available at <http://insecure.org/stf/smashstack.html>.

Aleph One did not invent the stack overflow; knowledge and exploitation of stack overflows had been passed around for a decade or longer before "Smashing the Stack" was released. Stack overflows have theoretically been around for at least as long as the C language and exploitation of these vulnerabilities has occurred regularly for well over 25 years. Even though they are likely the best understood and most publicly documented class of vulnerability, stack overflow vulnerabilities remain generally prevalent in software produced today. Check your favorite security news list; it's likely that a stack overflow vulnerability is being reported even as you read this chapter.

Buffers

A *buffer* is defined as a limited, contiguously allocated set of memory. The most common buffer in C is an *array*. The introductory material in this chapter focuses on arrays.

Stack overflows are possible because no inherent bounds-checking exists on buffers in the C or C++ languages. In other words, the C language and its derivatives do not have a built-in function to ensure that data being copied into a buffer will not be larger than the buffer can hold.

Consequently, if the person designing the program has not explicitly coded the program to check for oversized input, it is possible for data to fill a buffer, and if that data is large enough, to continue to write past the end of the buffer. As you will see in this chapter, all sorts of crazy things start happening once you write past the end of a buffer. Take a look at this extremely simple example that illustrates how C has no bounds-checking on buffers. (Remember, you can find this and many other code fragments and programs on *The Shellcoder's Handbook* Web site, <http://www.wiley.com/go/shellcodershandbook>.)

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

int main ()
{
    int array[5] = {1, 2, 3, 4, 5};

    printf("%d\n", array[5] );
}
```

In this example, we have created an array in C. The array, named `array`, is five elements long. We have made a novice C programmer mistake here, in that we forgot that an array of size five begins with element zero, `array[0]`, and ends with element four, `array[4]`. We tried to read what we thought was the fifth element of the array, but we were really reading beyond the array, into the “sixth” element. The `gcc` compiler elicits no errors, but when we run this code, we get unexpected results:

```
shellcoders@debian:~/chapter_2$ cc buffer.c
shellcoders@debian:~/chapter_2$ ./a.out
134513712
```

This example shows how easy it is to read past the end of a buffer; C provides no built-in protection. What about writing past the end of a buffer? This must be possible as well. Let's intentionally try to write way past the buffer and see what happens:

```
int main ()
{
```

```
int array[5];
int i;

for (i = 0; i <= 255; i++ )
{
    array[i] = 10;
}
}
```

Again, our compiler gives us no warnings or errors. But, when we execute this program, it crashes:

```
shellcoders@debian:~/chapter_2$ cc buffer2.c
shellcoders@debian:~/chapter_2$ ./a.out
Segmentation fault (core dumped)
```

As you might already know from experience, when a programmer creates a buffer that has the potential to be overflowed and then compiles and runs the code, the program often crashes or does not function as expected. The programmer then goes back through the code, discovers where he or she made a mistake, and fixes the bug. Let's have a peek at the core dump in `gdb`:

```
shellcoders@debian:~/chapter_2$ gdb -q -c core
Program terminated with signal 11, Segmentation fault.
#0  0x0000000a in ?? ()
(gdb)
```

Interestingly, we see that the program was executing address `0x0000000a`—or 10 in decimal—when it crashed. More on this later in this chapter.

So, what if user input is copied into a buffer? Or, what if a program expects input from another program that can be emulated by a person, such as a TCP/IP network-aware client?

If the programmer designs code that copies user input into a buffer, it may be possible for a user to intentionally place more input into a buffer than it can hold. This can have a number of different consequences, everything from crashing the program to forcing the program to execute user-supplied instructions. These are the situations we are chiefly concerned with, but before we get to control of execution, we first need to look at how overflowing a buffer stored on the stack works from a memory management perspective.

The Stack

As discussed in Chapter 1, the stack is a LIFO data structure. Much like a stack of plates in a cafeteria, the last element placed on the stack is the first element that must be removed. The boundary of the stack is defined by the extended

stack pointer (*ESP*) register, which points to the top of the stack. Stack-specific instructions, *PUSH* and *POP*, use *ESP* to know where the stack is in memory. In most architectures, especially IA32, on which this chapter is focused, *ESP* points to the last address used by the stack. In other implementations, it points to the first free address.

Data is placed onto the stack using the *PUSH* instruction; it is removed from the stack using the *POP* instruction. These instructions are highly optimized and efficient at moving data onto and off of the stack. Let's execute two *PUSH* instructions and see how the stack changes:

```
push 1
push addr var
```

These two instructions will first place the value 1 on the stack, then place the address of variable *VAR* on top of it. The stack will look like that shown in Figure 2-1.

Address Value	
643410h Address of variable VAR	← <i>ESP</i> points to this address
643414h 1	
643418h	

Figure 2-1: PUSHing values onto the stack

The *ESP* register will point to the top of the stack, address 643410h. Values are pushed onto the stack in the order of execution, so we have the value 1 pushed on first, and then the address of variable *VAR*. When a *PUSH* instruction is executed, *ESP* is decremented by four, and the dword is written to the new address stored in the *ESP* register.

Once we have put something on the stack, inevitably, we will want to retrieve it—this is done with the *POP* instruction. Using the same example, let's retrieve our data and address from the stack:

```
pop eax
pop ebx
```

First, we load the value at the top of the stack (where *ESP* is pointing) into *EAX*. Next, we repeat the *POP* instruction, but copy the data into *EBX*. The stack now looks like that shown in Figure 2-2.

As you may have already guessed, the *POP* instruction only changes the value of *ESP*—it does not write or erase data from the stack. Rather, *POP* writes

data to the operand, in this case first writing the address of variable `VAR` to `EAX` and then writing the value `1` to `EBX`.

Address Value
643410h Address of variable VAR
643414h 1
643418h

← ESP points to this address

Figure 2-2: POPing values from the stack

Another relevant register to the stack is `EBP`. The `EBP` register is usually used to calculate an address relative to another address, sometimes called a *frame pointer*. Although it can be used as a general-purpose register, `EBP` has historically been used for working with the stack. For example, the following instruction makes use of `EBP` as an index:

```
mov eax, [ebp+10h]
```

This instruction will move a dword from 16 bytes (10 in hex) down the stack (remember, the stack grows toward lower-numbered addresses) into `EAX`.

Functions and the Stack

The stack's primary purpose is to make the use of functions more efficient. From a low-level perspective, a function alters the flow of control of a program, so that an instruction or group of instructions can be executed independently from the rest of the program. More important, when a function has completed executing its instructions, it returns control to the original function caller. This concept of functions is most efficiently implemented with the use of the stack.

Take a look at a simple C function and how the stack is used by the function:

```
void function(int a, int b)
{
    int array[5];
}

main()
{
    function(1,2);

    printf("This is where the return address points");
}
```

In this example, instructions in `main` are executed until a function call is encountered. The consecutive execution of the program now needs to be interrupted, and the instructions in `function` need to be executed. The first step is to push the arguments for `function`, `a` and `b`, backward onto the stack. When the arguments are placed onto the stack, the function is called, placing the return address, or `RET`, onto the stack. `RET` is the address stored in the instruction pointer (`EIP`) at the time `function` is called. `RET` is the location at which to continue execution when the function has completed, so the rest of the program can execute. In this example, the address of the `printf("This is where the return address points");` instruction will be pushed onto the stack.

Before any `function` instructions can be executed, the prolog is executed. In essence, the prolog stores some values onto the stack so that the function can execute cleanly. The current value of `EBP` is pushed onto the stack, because the value of `EBP` must be changed in order to reference values on the stack. When the function has completed, we will need this stored value of `EBP` in order to calculate address locations in `main`. Once `EBP` is stored on the stack, we are free to copy the current stack pointer (`ESP`) into `EBP`. Now we can easily reference addresses local to the stack.

The last thing the prolog does is to calculate the address space required for the variables local to `function` and reserve this space on the stack. Subtracting the size of the variables from `ESP` reserves the required space. Finally, the variables local to `function`, in this case simply `array`, are pushed onto the stack. Figure 2-3 represents how the stack looks at this point.

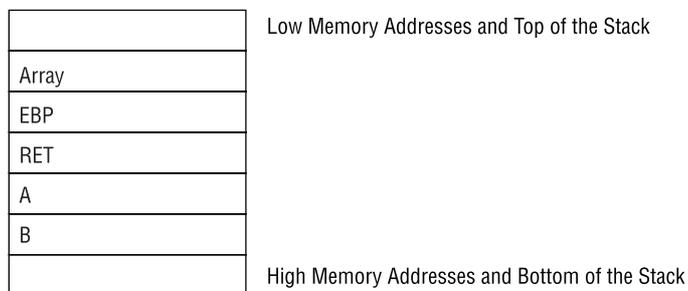


Figure 2-3: Visual representation of the stack after a function has been called

Now you should have a good understanding of how a function works with the stack. Let's get a little more in-depth and look at what is going on from an assembly perspective. Compile our simple C function with the following command:

```
shellcoders@debian:~/chapter_2$ cc -mpreferred-stack-boundary=2 -ggdb
function.c -o function
```