CONTENTS IN DETAIL

FOREWORD by Herbert Bos xvii

PREFACE xxi

ACKNOWLEDGMENTS xxiii

INTRODUCTION 1

What Is Binary Analysis, and Why Do You Need It? 2
What Makes Binary Analysis Challenging? 3
Who Should Read This Book? 4
What's in This Book? 4
How to Use This Book 5
Instruction Set Architecture 6
Assembly Syntax 6
Binary Format and Development Platform 6
Code Samples and Virtual Machine 7
Exercises 8

PART I: BINARY FORMATS

1
ANATOMY OF A BINARY 11

1.1 The C Compilation Process 12
1.1.1 The Preprocessing Phase 12
1.1.2 The Compilation Phase 14
1.1.3 The Assembly Phase 16
1.1.4 The Linking Phase 17
1.2 Symbols and Stripped Binaries 18
1.2.1 Viewing Symbolic Information 18
1.2.2 Another Binary Turns to the Dark Side: Stripping a Binary 20
1.3 Disassembling a Binary 21
1.3.1 Looking Inside an Object File 21
1.3.2 Examining a Complete Binary Executable 23
1.4 Loading and Executing a Binary 27
1.5 Summary 29
Exercises 29

2
THE ELF FORMAT 31

2.1 The Executable Header 33
2.1.1 The e_ident Array 34
2.1.2 The e_type, e_machine, and e_version Fields 35
2.1.3 The e_entry Field 36
2.1.4 The e_phoff and e_shoff Fields 36
2.1.5 The e_flags Field 36
2.1.6 The e_ehsize Field 37
2.1.7 The e_*entsize and e_*num Fields 37
2.1.8 The e_shstrndx Field 37
2.2 Section Headers 38
2.2.1 The sh_name Field 39
2.2.2 The sh_type Field 39
2.2.3 The sh_flags Field 40
2.2.4 The sh_addr, sh_offset, and sh_size Fields 40
2.2.5 The sh_link Field 40
2.2.6 The sh_info Field 41
2.2.7 The sh_addralign Field 41
2.2.8 The sh_entsize Field 41
2.3 Sections 41
2.3.1 The .init and .fini Sections 43
2.3.2 The .text Section 43
2.3.3 The .bss, .data, and .rodata Sections 44
2.3.4 Lazy Binding and the .plt, .got, and .got.plt Sections 45
2.3.5 The .rel.* and .rela.* Sections 48
2.3.6 The .dynamic Section 50
2.3.7 The .init_array and .fini_array Sections 51
2.3.8 The .shstrtab, .symtab, .strtab, .dynsym, and .dynstr Sections 52
2.4 Program Headers 52
2.4.1 The p_type Field 54
2.4.2 The p_flags Field 54
2.4.3 The p_offset, p_vaddr, p_paddr, p_filesz, and p_memsz Fields 54
2.4.4 The p_align Field 55
2.5 Summary 55
Exercises 55

3
THE PE FORMAT: A BRIEF INTRODUCTION 57

3.1 The MS-DOS Header and MS-DOS Stub 58
3.2 The PE Signature, File Header, and Optional Header 58
3.2.1 The PE Signature 61
3.2.2 The PE File Header 61
3.2.3 The PE Optional Header 62
3.3 The Section Header Table 62

3.4 Sections 63
3.4.1 The .edata and .idata Sections 64
3.4.2 Padding in PE Code Sections 64
3.5 Summary 65
Exercises 65

4
BUILDING A BINARY LOADER USING LIBBFD 67

4.1 What Is libbfd? 68
4.2 A Simple Binary-Loading Interface 68
4.2.1 The Binary Class 71
4.2.2 The Section Class 71
4.2.3 The Symbol Class 71
4.3 Implementing the Binary Loader 72
4.3.1 Initializing libbfd and Opening a Binary 73
4.3.2 Parsing Basic Binary Properties 75
4.3.3 Loading Symbols 78
4.3.4 Loading Sections 81
4.4 Testing the Binary Loader 83
4.5 Summary 85
Exercises 86

PART II: BINARY ANALYSIS FUNDAMENTALS

5
BASIC BINARY ANALYSIS IN LINUX 89

5.1 Resolving Identity Crises Using file 90
5.2 Using ldd to Explore Dependencies 93
5.3 Viewing File Contents with xxd 94
5.4 Parsing the Extracted ELF with readelf 96
5.5 Parsing Symbols with nm 99
5.6 Looking for Hints with strings 102
5.7 Tracing System Calls and Library Calls with strace and ltrace 104
5.8 Examining Instruction-Level Behavior Using objdump 109
5.9 Dumping a Dynamic String Buffer Using gdb 111
5.10 Summary 113
Exercise 113

6
DISASSEMBLY AND BINARY ANALYSIS FUNDAMENTALS 115

6.1 Static Disassembly 116
6.1.1 Linear Disassembly 117
6.1.2 Recursive Disassembly 118

6.2 Dynamic Disassembly 122
6.2.1 Example: Tracing a Binary Execution with gdb 122
6.2.2 Code Coverage Strategies 125
6.3 Structuring Disassembled Code and Data 129
6.3.1 Structuring Code 129
6.3.2 Structuring Data 136
6.3.3 Decompilation 138
6.3.4 Intermediate Representations 139
6.4 Fundamental Analysis Methods 141
6.4.1 Binary Analysis Properties 142
6.4.2 Control-Flow Analysis 146
6.4.3 Data-Flow Analysis 148
6.5 Effects of Compiler Settings on Disassembly 152
6.6 Summary 153
Exercises 153

7
SIMPLE CODE INJECTION TECHNIQUES FOR ELF 155

7.1 Bare-Metal Binary Modification Using Hex Editing 155
7.1.1 Observing an Off-by-One Bug in Action 156
7.1.2 Fixing the Off-by-One Bug 159
7.2 Modifying Shared Library Behavior Using LD_PRELOAD 163
7.2.1 A Heap Overflow Vulnerability 163
7.2.2 Detecting the Heap Overflow 165
7.3 Injecting a Code Section 169
7.3.1 Injecting an ELF Section: A High-Level Overview 169
7.3.2 Using elfinject to Inject an ELF Section 171
7.4 Calling Injected Code 175
7.4.1 Entry Point Modification 176
7.4.2 Hijacking Constructors and Destructors 179
7.4.3 Hijacking GOT Entries 182
7.4.4 Hijacking PLT Entries 185
7.4.5 Redirecting Direct and Indirect Calls 186
7.5 Summary 187
Exercises 187

PART III: ADVANCED BINARY ANALYSIS

8
CUSTOMIZING DISASSEMBLY 191

8.1 Why Write a Custom Disassembly Pass? 192
8.1.1 A Case for Custom Disassembly: Obfuscated Code 192
8.1.2 Other Reasons to Write a Custom Disassembler 195

8.2 Introduction to Capstone 196
8.2.1 Installing Capstone 196
8.2.2 Linear Disassembly with Capstone 198
8.2.3 Exploring the Capstone C API 203
8.2.4 Recursive Disassembly with Capstone 204
8.3 Implementing a ROP Gadget Scanner 213
8.3.1 Introduction to Return-Oriented Programming 213
8.3.2 Finding ROP Gadgets 215
8.4 Summary 221
Exercises 221

9
BINARY INSTRUMENTATION 223

9.1 What Is Binary Instrumentation? 224
9.1.1 Binary Instrumentation APIs 224
9.1.2 Static vs. Dynamic Binary Instrumentation 225
9.2 Static Binary Instrumentation 226
9.2.1 The int 3 Approach 227
9.2.2 The Trampoline Approach 228
9.3 Dynamic Binary Instrumentation 233
9.3.1 Architecture of a DBI System 233
9.3.2 Introduction to Pin 235
9.4 Profiling with Pin 237
9.4.1 The Profiler's Data Structures and Setup Code 237
9.4.2 Parsing Function Symbols 240
9.4.3 Instrumenting Basic Blocks 241
9.4.4 Instrumenting Control Flow Instructions 243
9.4.5 Counting Instructions, Control Transfers, and Syscalls 246
9.4.6 Testing the Profiler 247
9.5 Automatic Binary Unpacking with Pin 251
9.5.1 Introduction to Executable Packers 251
9.5.2 The Unpacker's Data Structures and Setup Code 253
9.5.3 Instrumenting Memory Writes 255
9.5.4 Instrumenting Control-Flow Instructions 256
9.5.5 Tracking Memory Writes 256
9.5.6 Detecting the Original Entry Point and Dumping the Unpacked Binary 258
9.5.7 Testing the Unpacker 259
9.6 Summary 263
Exercises 264

10
PRINCIPLES OF DYNAMIC TAINT ANALYSIS 265

10.1 What Is DTA? 266
10.2 DTA in Three Steps: Taint Sources, Taint Sinks, and Taint Propagation 266
    10.2.1 Defining Taint Sources 266
    10.2.2 Defining Taint Sinks 267
    10.2.3 Tracking Taint Propagation 267
10.3 Using DTA to Detect the Heartbleed Bug 268
    10.3.1 A Brief Overview of the Heartbleed Vulnerability 268
    10.3.2 Detecting Heartbleed Through Tainting 269
10.4 DTA Design Factors: Taint Granularity, Taint Colors, and Taint Policies 271
    10.4.1 Taint Granularity 271
    10.4.2 Taint Colors 272
    10.4.3 Taint Propagation Policies 273
    10.4.4 Overtainting and Undertainting 274
    10.4.5 Control Dependencies 275
    10.4.6 Shadow Memory 276
10.5 Summary 278
Exercise 278

11
PRACTICAL DYNAMIC TAINT ANALYSIS WITH LIBDFT 279

11.1 Introducing libdft 279
    11.1.1 Internals of libdft 280
    11.1.2 Taint Policy 282
11.2 Using DTA to Detect Remote Control-Hijacking 283
    11.2.1 Checking Taint Information 286
    11.2.2 Taint Sources: Tainting Received Bytes 288
    11.2.3 Taint Sinks: Checking execve Arguments 290
    11.2.4 Detecting a Control-Flow Hijacking Attempt 291
11.3 Circumventing DTA with Implicit Flows 296
11.4 A DTA-Based Data Exfiltration Detector 297
    11.4.1 Taint Sources: Tracking Taint for Open Files 299
    11.4.2 Taint Sinks: Monitoring Network Sends for Data Exfiltration 303
    11.4.3 Detecting a Data Exfiltration Attempt 304
11.5 Summary 307
Exercise 307

12
PRINCIPLES OF SYMBOLIC EXECUTION 309

12.1 An Overview of Symbolic Execution 309
    12.1.1 Symbolic vs. Concrete Execution 310
    12.1.2 Variants and Limitations of Symbolic Execution 313
    12.1.3 Increasing the Scalability of Symbolic Execution 319

12.2 Constraint Solving with Z3 321
12.2.1 Proving Reachability of an Instruction 322
12.2.2 Proving Unreachability of an Instruction 325
12.2.3 Proving Validity of a Formula 325
12.2.4 Simplifying Expressions 327
12.2.5 Modeling Constraints for Machine Code with Bitvectors 327
12.2.6 Solving an Opaque Predicate Over Bitvectors 329
12.3 Summary 330
Exercises 330

13
PRACTICAL SYMBOLIC EXECUTION WITH TRITON 333

13.1 Introduction to Triton 334
13.2 Maintaining Symbolic State with Abstract Syntax Trees 335
13.3 Backward Slicing with Triton 337
13.3.1 Triton Header Files and Configuring Triton 340
13.3.2 The Symbolic Configuration File 340
13.3.3 Emulating Instructions 342
13.3.4 Setting Triton's Architecture 343
13.3.5 Computing the Backward Slice 344
13.4 Using Triton to Increase Code Coverage 346
13.4.1 Creating Symbolic Variables 348
13.4.2 Finding a Model for a New Path 348
13.4.3 Testing the Code Coverage Tool 352
13.5 Automatically Exploiting a Vulnerability 355
13.5.1 The Vulnerable Program 356
13.5.2 Finding the Address of the Vulnerable Call Site 359
13.5.3 Building the Exploit Generator 361
13.5.4 Getting a Root Shell 367
13.6 Summary 370
Exercise 370

PART IV: APPENDIXES

A
A CRASH COURSE ON X86 ASSEMBLY 373

A.1 Layout of an Assembly Program 374
A.1.1 Assembly Instructions, Directives, Labels, and Comments 374
A.1.2 Separation Between Code and Data 375
A.1.3 AT&T vs. Intel Syntax 376
A.2 Structure of an x86 Instruction 376
A.2.1 Assembly-Level Representation of x86 Instructions 376
A.2.2 Machine-Level Structure of x86 Instructions 376
A.2.3 Register Operands 377

A.2.4 Memory Operands 379
A.2.5 Immediates 380
A.3 Common x86 Instructions 380
A.3.1 Comparing Operands and Setting Status Flags 382
A.3.2 Implementing System Calls 382
A.3.3 Implementing Conditional Jumps 382
A.3.4 Loading Memory Addresses 383
A.4 Common Code Constructs in Assembly 383
A.4.1 The Stack 383
A.4.2 Function Calls and Function Frames 384
A.4.3 Conditional Branches 388
A.4.4 Loops 389

B
IMPLEMENTING PT_NOTE OVERWRITING USING LIBELF 391

B.1 Required Headers 392
B.2 Data Structures Used in elfinject 392
B.3 Initializing libelf 393
B.4 Getting the Executable Header 397
B.5 Finding the PT_NOTE Segment 398
B.6 Injecting the Code Bytes 399
B.7 Aligning the Load Address for the Injected Section 400
B.8 Overwriting the .note.ABI-tag Section Header 401
B.9 Setting the Name of the Injected Section 406
B.10 Overwriting the PT_NOTE Program Header 408
B.11 Modifying the Entry Point 410

C
LIST OF BINARY ANALYSIS TOOLS 413

C.1 Disassemblers 413
C.2 Debuggers 415
C.3 Disassembly Frameworks 415
C.4 Binary Analysis Frameworks 416

D
FURTHER READING 417

D.1 Standards and References 417
D.2 Papers and Articles 418
D.3 Books 420

INDEX 421