X64 Reverse Engineering Exercises

Easy Exercises (1-2 hours each)

Basic Binary Analysis

1. **simple_xor** - Given a binary that XORs input with a constant key, find the key by analyzing the disassembly. Binary reads string from stdin, XORs each byte with key, prints result.
2. **password_check** - Reverse a binary with hardcoded password check (simple strcmp). Find the password by examining strings and control flow.
3. **flag_decoder** - Binary contains encoded flag string. Find the decoding algorithm (simple Caesar cipher or ROT13) and recover the flag.
4. **basic_crackme** - Binary checks serial number using simple arithmetic (e.g., sum of digits must equal constant). Find valid serial by analyzing the algorithm.
5. **stack_strings** - Binary constructs strings on stack byte-by-byte to hide them. Extract the hidden strings from the binary.

Control Flow Analysis

6. **if_else_maze** - Binary with nested if-else statements. Map the control flow and find input that reaches "success" branch.
7. **switch_table** - Analyze binary using jump table for switch statement. Determine all valid case values and their outputs.
8. **loop_counter** - Binary performs calculation in a loop. Determine loop count and final result without running (static analysis only).

Function Analysis

9. **function_calls** - Binary with multiple functions calling each other. Build call graph and identify the main algorithm.
10. **return_value** - Analyze function that performs computation and returns value. Determine what value is returned for given inputs by reading assembly.

Anti-Analysis - Basic

11. **stripped_binary** - Binary with symbols stripped. Identify main function and key functions by analyzing entry point and calling conventions.
12. **upx_packed** - Unpack UPX-packed binary and analyze the original program.

Data Structure Recognition

13. **array_access** - Identify array indexing patterns and determine array size and element type from memory access patterns.
14. **struct_layout** - Analyze code accessing struct members. Determine struct field offsets, types, and layout.

Patching

15. **nop_check** - Binary has annoying delay or check. Patch it out by replacing instructions with NOPs.
16. **jmp_patch** - Modify conditional jump to unconditional jump to bypass license check.

Format String Exploits

17. **format_string_leak** - Binary has format string vulnerability. Find what memory addresses leak when you provide specific format string.

Number Theory & Crypto - Basic

18. **simple_hash_reverse** - Binary computes simple hash of input. Reverse the algorithm and find collision or preimage.
19. **checksum_bypass** - Binary validates input using simple checksum (XOR or sum). Find input that passes validation.

Syscall Analysis

20. **syscall_tracer** - Analyze binary and list all system calls it makes without running it (static analysis).
21. **file_operations** - Binary performs file operations. Determine what files it opens, reads, writes by analyzing syscalls.

Code Obfuscation - Basic

22. **opaque_predicates** - Binary uses opaque predicates (always true/false conditions). Identify them and simplify control flow.
23. **dead_code** - Binary contains dead code branches. Identify which branches are unreachable.

ARM Assembly

24. **arm_basic** - Simple ARM64 binary with basic arithmetic. Analyze and determine output for given input.
25. **thumb_mode** - Analyze ARM binary switching between ARM and Thumb modes. Identify mode switches and follow execution.

Medium Exercises (3-5 hours each)

Obfuscation & Anti-Debug

1. **anti_debug_ptrace** - Binary checks for debugger using ptrace. Find all anti-debug checks and patch them to allow debugging.
2. **timing_checks** - Binary uses RDTSC to detect debugger by measuring execution time. Find timing checks and bypass them.
3. **control_flow_flattening** - Binary uses control flow flattening obfuscation (dispatcher pattern). De-obfuscate and recover original control flow graph.

Cryptography

4. **custom_cipher** - Binary implements custom symmetric cipher. Reverse the algorithm and write decoder. Test with known plaintext-ciphertext pairs.
5. **tea_variant** - Binary implements variant of TEA (Tiny Encryption Algorithm) with modified constants. Extract the key and decrypt provided ciphertext.
6. **rc4_implementation** - Identify that binary uses RC4 stream cipher. Extract the key from the binary and decrypt messages.

Virtual Machines

7. **simple_vm** - Binary implements simple bytecode VM with custom instruction set. Reverse the instruction set and write disassembler for the bytecode.
8. **vm_protected_code** - Part of binary's logic is protected by simple VM. Analyze VM, extract bytecode, and understand protected algorithm.

Malware Analysis - Basic

9. **dropper_analysis** - Analyze dropper that extracts embedded executable. Find the embedded payload and extraction algorithm.
10. **c2_protocol** - Binary communicates with C2 server using custom protocol. Reverse the protocol structure and message format (no network required, analyze code only).

Advanced Patching

11. **license_keygen** - Reverse complex license validation algorithm (involving CRC, hashing, bit operations). Write keygen that produces valid licenses.
12. **trial_extension** - Binary has 30-day trial with encrypted timestamp. Find encryption, modify stored data or patch checks to extend trial indefinitely.

Binary Formats

13. **custom_format_parser** - Binary parses custom binary format. Reverse the format specification and write parser in C/Python.
14. **elf_modifier** - Analyze how binary modifies its own ELF headers at runtime. Understand the self-modification technique.

Concurrency

15. **race_condition** - Multi-threaded binary with race condition vulnerability. Identify the race window and shared resource by analyzing thread synchronization.

Hard Exercises (Multiple evenings)

Advanced Obfuscation

1. **ollvm_protected** - Binary protected with OLLVM (control flow flattening + bogus control flow + instruction substitution). De-obfuscate at least the main algorithm and recover original logic. Use tools like angr, Ghidra, or manual analysis.
2. **virtualization_obfuscation** - Binary protected with commercial virtualization obfuscator (VMProtect-style). Analyze VM architecture, reverse instruction handlers, and extract original algorithm for at least one protected function.
3. **mixed_boolean_arithmetic** - Binary uses MBA (Mixed Boolean-Arithmetic) obfuscation. Simplify obfuscated expressions to recover original arithmetic operations. Write pattern-based deobfuscator.

Kernel/Driver Analysis

4. **kernel_module** - Analyze Linux kernel module (.ko file). Identify what kernel functions it hooks, what it monitors, or what functionality it adds. No need to load module, pure static analysis.
5. **rootkit_detection** - Analyze kernel-mode rootkit that hides processes/files. Identify hooking mechanism and hidden artifacts by reverse engineering the module.

Advanced Cryptography

6. **aes_custom_sbox** - Binary implements AES with custom S-box. Extract the custom S-box and modified round constants. Decrypt provided ciphertext.
7. **rsa_impl_attack** - Binary implements RSA with small key (512-bit) or weak parameters. Extract public/private exponents and modulus, identify weakness (e.g., shared factors), break the encryption.

Exploitation Development

8. **buffer_overflow_exploit** - Binary has stack buffer overflow. Reverse to find vulnerability, calculate offset, and develop working exploit (write shellcode + ROP chain if NX enabled) to spawn shell.
9. **format_string_exploit** - Binary has format string vulnerability. Develop exploit to leak addresses, overwrite GOT entry, and achieve code execution.

Protocol Reversing

10. **network_protocol** - Binary implements proprietary network protocol. Reverse message structure, authentication, encryption. Write client that can communicate using the protocol. Can analyze pcap files or binary code directly.

Exercise Formats & Testing Setup

Binary Types

- **ELF64 executables** (x86-64 Linux) - primary format
- **Stripped binaries** (no symbols) - medium/hard exercises
- **Statically linked** - some exercises to avoid libc dependency analysis
- **PIE/non-PIE** - various configurations
- **ARM64 binaries** - for ARM-specific exercises (run with qemu-aarch64)

Tools You'll Need

Disassemblers/Decompilers:

- Ghidra (free, excellent decompiler)
- IDA Free (limited but powerful)
- radare2/Cutter (open source)
- objdump (basic disassembly)
- Binary Ninja (commercial, but has demo)

Debuggers:

- GDB with pwndbg/GEF/peda

- radare2 debugger
- EDB (Evan's Debugger)

Analysis Tools:

- ltrace/strace (library/system call tracing)
- strings, file, readelf
- binwalk (for embedded files)
- UPX (packer/unpacker)
- angr (symbolic execution)
- ROPgadget, ropper (ROP chain building)

Binary Modification:

- patchelf (modify ELF properties)
- xxd, hexedit (hex editing)
- ld (relinking)
- gcc/nasm (recompiling modified code)

Test Setup Workflow

For each exercise you'll receive:

1. **Binary file** (ELF64 or ARM64)
2. **Challenge description** (what to find/achieve)
3. **Success criteria** (e.g., "extract the flag", "find valid key", "write working exploit")
4. **Hints file** (optional, for when stuck)

Sample directory structure:

```
exercise_01_simple_xor/
├── challenge (ELF64 binary)
├── README.md (description)
├── input_sample.txt (if needed)
└── hints.txt (optional)
```

Verification Methods

Easy exercises:

- Find specific string/value
- Produce valid input that satisfies check
- Successfully patch binary to bypass check

Medium exercises:

- Write keygen that produces valid keys
- Decrypt provided ciphertext
- Write disassembler/decompiler for VM
- Document protocol/format specification

Hard exercises:

- De-obfuscate and explain algorithm
- Working exploit (validated in controlled environment)
- Break cryptography and decrypt message
- Reverse complex protocol and write compatible client

Scoring/Validation

Automated testing possible for:

- Keygens (test generated keys against original binary)
- Decoders (compare with known plaintext)
- Format parsers (parse test files)
- Exploits (verify shell spawned or arbitrary code executed)

Manual review needed for:

- De-obfuscation quality
- Documentation completeness
- Code analysis write-ups
- Algorithm explanations

Difficulty Calibration

Easy (1-2h):

- Single technique/concept
- Minimal obfuscation
- Clear success condition
- Standard tools sufficient

Medium (3-5h):

- Multiple techniques combined
- Some obfuscation/anti-analysis
- Requires scripting/automation
- May need custom tools

Hard (2-3 evenings):

- Heavy obfuscation or complex algorithms
- Requires advanced analysis techniques
- Significant development (exploits, deobfuscators)
- Research may be needed
- Tool development required

Additional Resources

Practice platforms with similar exercises:

- Crackmes.one (crackme collection)
- ReverseEngineering on Reddit (weekly challenges)
- PicoCTF, HackTheBox (CTF platforms with RE challenges)
- Malware analysis reports (real-world examples)
- Flare-On challenge archives

Learning progression:

1. Start with easy exercises to learn tools
2. Move to medium to practice techniques
3. Hard exercises to master advanced topics
4. Optionally create your own obfuscated binaries as practice

This list provides comprehensive coverage of reverse engineering skills from basic static analysis through advanced de-obfuscation and exploitation, all testable on your x64 Debian system with AMD CPU.