

Binary & Cryptographic Analysis

Reverse engineering binaries, breaking weak crypto, and uncovering hidden data — with AI as your analysis partner.

Reverse Engineering

Cryptanalysis

Steganography

AI-Assisted

CTF

⚠ Educational Use Only

Binary analysis and cryptographic attack techniques are for authorized security research and CTF competitions only. Creating or distributing malware is illegal.

Learning Objectives

1. **Understand ELF and PE binary formats** and how to triage unknown executables
 2. **Use disassemblers and decompilers** (Ghidra, radare2) to analyze compiled code
 3. **Leverage AI to interpret assembly** and identify vulnerability patterns
 4. **Identify and exploit cryptographic weaknesses** in CTF challenges
 5. **Apply hash cracking strategies** using AI-generated wordlists
 6. **Detect steganographic content** hidden in files using AI anomaly detection
-

Why Binary Analysis Matters

In CTFs and real-world red teaming, you rarely have source code. Binaries are the artifact you must understand.

CTF Context

Reverse engineering challenges are worth the most points. Crypto challenges reward pattern recognition and mathematical insight.

Red Team Context

Malware analysis, firmware extraction, proprietary protocol reversing, and breaking custom encryption in enterprise software.

The Analysis Toolchain

Tool	Purpose	Key Strength
Ghidra	Disassembler & decompiler	Free, powerful decompilation to pseudo-C
radare2 / rizin	CLI reverse engineering	Scriptable, fast triage
CyberChef	Data transformation	Chain encodings, crypto, compression
hashcat / John	Hash cracking	GPU-accelerated, rule-based
binwalk	Firmware / file analysis	Embedded file extraction
Claude / AI	Analysis companion	Pattern recognition, code explanation

ELF File Format (Linux Binaries)

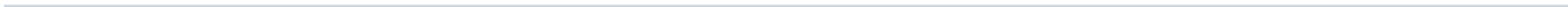
The **Executable and Linkable Format** is the standard binary format on Linux/Unix systems.

```
# Quick triage of an unknown binary
$ file mystery_binary
mystery_binary: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
not stripped

$ readelf -h mystery_binary | head -12
ELF Header:
  Magic:      7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                                ELF64
  Data:                                  2's complement, little endian
  Type:                                  EXEC (Executable file)
  Machine:                                Advanced Micro Devices X86-64
  Entry point address:                   0x401060
```

Key ELF Sections

`.text` — executable code | `.data` — initialized globals | `.bss` — uninitialized data |
`.rodata` — read-only strings | `.plt/.got` — dynamic linking



PE File Format (Windows Binaries)

The **Portable Executable** format is used by Windows for .exe, .dll, and .sys files.

```
# PE file structure (hex dump of header)
Offset | Hex                                     | ASCII
-----|-----|-----
00000000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff  MZ.....
0000000e  00 00 b8 00 00 00 00 00 00 00 40 00 00 00  .....@...
00000080  50 45 00 00 4c 01 05 00                    PE..L...
```

- **MZ header** — DOS stub, magic bytes `4D 5A`
- **PE signature** — `50 45 00 00` at offset pointed by e_lfanew
- **Sections:** `.text`, `.rdata`, `.data`, `.rsrc` (resources)
- **Import Table** — lists DLLs and API functions the binary calls

First Steps: Binary Triage

Before diving into disassembly, gather intelligence about the binary.

```
# Step 1: Identify the file
$ file challenge_bin

# Step 2: Check for readable strings
$ strings challenge_bin | grep -i "flag\|password\|key\|secret"
Enter password:
Wrong! Try again.
Congratulations!
flag{

# Step 3: Check linked libraries
$ ldd challenge_bin
libcrypto.so.1.1 => /usr/lib/x86_64-linux-gnu/libcrypto.so.1.1

# Step 4: Check symbols (if not stripped)
```

Notes: The strings command alone solves a surprising number of CTF reversing challenges. Always start simple.

Disassembly Basics: x86-64

Understanding common assembly patterns is essential for reverse engineering.

```
; Typical password check function
check_password:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 0x20
    mov     QWORD PTR [rbp-0x18], rdi    ; user input pointer
    lea     rax, [rip+0x2e5a]           ; load hardcoded string
    mov     QWORD PTR [rbp-0x8], rax
    mov     rdx, QWORD PTR [rbp-0x18]
    mov     rax, QWORD PTR [rbp-0x8]
    mov     rsi, rdx
    mov     rdi, rax
    call    strcmp                     ; compare strings!
    test    eax, eax
    jne     .wrong                     : jump if not equal
```

Key Patterns to Recognize

`call strcmp` — string comparison | `test eax, eax / jne` — branch on result | `lea rax,`
`[rip+offset]` — load address of hardcoded data

Ghidra: Decompilation to Pseudo-C

Ghidra's decompiler converts assembly back to readable C-like code — much easier to analyze.

```
// Ghidra decompiler output for check_password
int check_password(char *user_input) {
    char *hardcoded = "s3cr3t_p@ss!";
    int result;

    result = strcmp(hardcoded, user_input);
    if (result == 0) {
        return 1;
    }
    return 0;
}

// Another common pattern: XOR-based "encryption"
void decrypt_flag(char *buf, int len) {
    int i;
```

Notes: In the decompiled output, the hardcoded password is immediately visible. The XOR decrypt function reveals a single-byte key of 0x42 — trivially reversible.

AI-Assisted Disassembly Analysis

Feed decompiled code or assembly to an LLM for rapid understanding of complex functions.

```
import anthropic

client = anthropic.Anthropic()

ghidra_output = ""
void FUN_00401256(char *param_1) {
    int iVar1;
    size_t sVar2;
    char local_48[32];
    int local_28;
    int local_24;

    sVar2 = strlen(param_1);
    local_28 = (int)sVar2;
    for (local_24 = 0; local_24 < local_28; local_24++) {
```

AI Insight

Claude identifies this as **ROT13** applied to uppercase letters. The encoded string

`SYNT{ebg13_vf_abg_rapelcgvba}` decodes to `FLAG{rot13_is_not_encryption}`.

radare2: CLI Reverse Engineering

radare2 is ideal for quick command-line triage without a GUI.

```
# Open binary in analysis mode
$ r2 -A challenge_bin

# List functions
[0x00401060]> afl
0x00401060    1 42      entry0
0x00401090    4 41    -> 34  fcn.00401090
0x004011c0    1 5      sym.check_password
0x00401210    3 95      sym.decrypt_flag
0x00401280    1 32      sym.main

# Disassemble a function
[0x00401060]> pdf @sym.check_password

# Find cross-references to a string
```

Binary Exploitation: Buffer Overflow

The classic vulnerability — writing past the end of a buffer to overwrite the return address or other critical data.

```
// Vulnerable function (decompiled from CTF binary)
void login(void) {
    char password[16];           // only 16 bytes allocated
    int is_admin = 0;           // sits right after buffer on stack

    printf("Enter password: ");
    gets(password);             // NO bounds checking!

    if (is_admin != 0) {
        printf("Admin access granted!\n");
        print_flag();           // overflow password to set is_admin
    } else if (strcmp(password, "guest") == 0) {
        printf("Guest access.\n");
    }
}
```

```
# Exploit: overflow the 16-byte buffer to set is_admin
$ python3 -c 'print("A"*16 + "\x01")' | ./challenge_bin
Admin access granted!
flag{stack_smash_success_42}
```

Stack Layout

[password: 16 bytes][is_admin: 4 bytes][saved RBP][return address] — Overflowing

password writes into is_admin.

Binary Exploitation: Format Strings

When user input is passed directly as a format string to `printf`, attackers can read and write arbitrary memory.

```
// Vulnerable code
void greet(char *name) {
    printf(name);          // WRONG: should be printf("%s", name)
}

// What the attacker sends:
// "%x %x %x %x %x %x %x %x"
```

```
# Leaking stack values
$ echo '%p %p %p %p %p %p' | ./vuln_binary
0x7ffd3a2b1000 0x64 0x7f2a8c001234 0x666c61677b666d 0x745f6c65616b7d 0xa

# Decode the hex values to ASCII
$ python3 -c "
import struct
vals = [0x666c61677b666d, 0x745f6c65616b7d]
for v in vals:
    print(struct.pack('<Q', v).decode('ascii', errors='ignore'), end='')
"
flag{fmt_leak}
```

Format String Specifiers for Exploitation

`%x` — leak hex values | `%p` — leak pointers | `%s` — read string at address | `%n` — write to memory (most dangerous)

Binary Exploitation: Use-After-Free

Accessing heap memory after it has been freed can allow an attacker to control program behavior.

```
// Simplified UAF pattern from a CTF challenge
typedef struct {
    void (*handler)(void);    // function pointer
    char data[64];
} Request;

Request *req = malloc(sizeof(Request));
req->handler = normal_handler;

free(req);           // memory freed but pointer not nulled

// Attacker allocates same-size chunk, overwrites handler
char *evil = malloc(sizeof(Request));
memcpy(evil, &win_function_addr, 8); // overwrite func ptr
```

AI Analysis Prompt

"This decompiled code uses malloc/free. Identify any use-after-free vulnerabilities and explain how to exploit them in a CTF context."

Cryptographic Analysis

From binary exploitation, we turn to **breaking weak cryptography** — the other major pillar of CTF challenges.

What We Will Cover

- Weak hash algorithms
- ECB mode vulnerabilities
- Padding oracle attacks
- Custom cipher weaknesses

AI Advantage

- Identifying cipher types from ciphertext
 - Spotting implementation flaws
 - Generating targeted wordlists
 - Explaining mathematical attacks
-

Cryptographic Weaknesses: Hashing

Many CTF challenges use deliberately weak or outdated hashing algorithms.

Algorithm	Status	Vulnerability
MD5	Broken	Collision attacks in seconds; rainbow tables widely available
SHA-1	Deprecated	Practical collision attacks demonstrated (SHattered, 2017)
DES	Broken	56-bit key; brute-forced in hours on modern hardware
RC4	Broken	Statistical biases in keystream
ROT13 / XOR	Not encryption	Trivially reversible; no key secrecy

```
# Cracking an MD5 hash found in a CTF challenge
$ echo -n "5d41402abc4b2a76b9719d911017c592" > hash.txt
$ hashcat -m 0 hash.txt /usr/share/wordlists/rockyou.txt
5d41402abc4b2a76b9719d911017c592:hello
```

ECB Mode: The Penguin Problem

Electronic Codebook (ECB) mode encrypts each block independently — identical plaintext blocks produce identical ciphertext blocks.

```
# Demonstrating ECB weakness
from Crypto.Cipher import AES
import binascii

key = b'YELLOW SUBMARINE' # 16-byte key

# ECB mode - INSECURE
cipher_ecb = AES.new(key, AES.MODE_ECB)

# Same plaintext block -> same ciphertext block
block1 = cipher_ecb.encrypt(b'AAAAAAAAAAAAAAAA')
block2 = cipher_ecb.encrypt(b'AAAAAAAAAAAAAAAA')
print(block1 == block2) # True! Information leaks

# CTF exploit: if you control part of the plaintext.
```

CTF Pattern: ECB Byte-at-a-Time

If an oracle encrypts `your_input + secret` in ECB mode, pad your input to push one unknown byte to the block boundary. Brute-force that byte by comparing ciphertext blocks.

Padding Oracle Attacks

When a server reveals whether CBC decryption produced valid padding, an attacker can decrypt the entire ciphertext without the key.

```
# Padding oracle attack concept
# PKCS#7 padding: last N bytes all have value N
# \x01, \x02\x02, \x03\x03\x03, etc.

def padding_oracle_attack(ciphertext, oracle_fn):
    """
    oracle_fn(modified_ct) returns True if padding is valid.
    By flipping bits in block N-1, we control the decrypted
    value of block N. When padding is valid, we've found
    the intermediate state byte.
    """
    blocks = [ciphertext[i:i+16] for i in range(0, len(ciphertext), 16)]
    plaintext = b""

    for block_idx in range(1, len(blocks)):
```

AI-Assisted Crypto Analysis

AI excels at identifying cipher types from ciphertext characteristics and spotting implementation flaws.

```
prompt = """Analyze this ciphertext found in a CTF challenge.  
Identify the cipher and suggest an attack:
```

```
Ciphertext (base64):
```

```
VGhpcyBpcyBiYXNlNjQgZW5jb2RlZA==
```

```
Hex values found in the binary:
```

```
key = 0x4B455931 (ASCII: "KEY1")
```

```
iv = 0x00000000
```

```
The encryption function XORs each byte of plaintext with  
the corresponding byte of the key, cycling the key."""
```

```
# AI response identifies:
```

```
# 1. The base64 decodes to "This is base64 encoded"
```

What AI Can Identify

Base64 (character set + padding) | **Hex encoding** | **Caesar/ROT** (letter frequency) | **XOR cipher** (repeated patterns) | **RSA** (large integers) | **AES-ECB** (block repetition)

Breaking Weak Crypto Implementations

Even strong algorithms can be broken when implemented incorrectly.

```
# CTF challenge: "custom encryption" that's just XOR
def custom_encrypt(plaintext, key):
    """The 'unbreakable' encryption from the challenge binary"""
    result = []
    for i, ch in enumerate(plaintext):
        result.append(chr(ord(ch) ^ key[i % len(key)]))
    return ''.join(result)

# Encrypted flag found in binary (hex):
encrypted_hex = "06 16 02 1b 56 13 05 07 03 16 1a 56 01 12 17 56 05 00 07 09"
encrypted = bytes.fromhex(encrypted_hex.replace(" ", ""))

# Known plaintext attack: we know flag starts with "flag{"
known_pt = b"flag{"
key_fragment = bytes(e ^ p for e, p in zip(encrypted[:5], known_pt))
```

CyberChef: The Crypto Swiss Army Knife

CyberChef lets you chain encoding, decoding, and crypto operations in a visual pipeline.

Common CTF Recipes

- **Multi-layer decode:** From Hex → From Base64 → ROT13 → flag
- **XOR brute-force:** XOR Brute Force (key length 1) with "flag" as crib
- **Magic mode:** Automatically detect and decode encoding layers

```
# Example: multi-layered CTF encoding
# Layer 1 (Hex): 52 6d 78 68 5a 33 74 6a 65 57 4a 6c 63 6d 4e 6f 5a 57 5a 39
# Layer 2 (ASCII): RmxhZ3tjeWJlcmNoZWZ9
# Layer 3 (Base64 decode): Flag{cyberchef}
```

CyberChef Recipe:

```
From_Hex('Space')
From_Base64()
```

Output: Flag{cyberchef}

Notes: CyberChef is available at gchq.github.io/CyberChef and also runs locally. The "Magic" operation auto-detects encodings — a CTF lifesaver.

Hash Cracking with AI-Generated Wordlists

AI can generate targeted wordlists based on context clues from the challenge.

```
# Ask AI to generate a contextual wordlist
prompt = """A CTF challenge about a space-themed company called
'NovaStar Labs' has a SHA-256 hash to crack. The hint says:
'The password is the CEO's pet project name.'

Generate 50 likely passwords combining space/nova/star themes
with common patterns (capitalize, numbers, special chars)."""

# AI generates targeted list:
wordlist = [
    "NovaStar2024", "novastar!", "StarNova#1", "NovaPrime",
    "SuperNova!", "NovaProject", "StarLabs123", "nova_star",
    "N0v4St4r", "NovaStar!", "ProjectNova", "DarkStar42",
    # ... more context-aware guesses
]
```

```
# Crack with AI-generated wordlist + rules
$ hashcat -m 1400 hash.txt ai_wordlist.txt -r /usr/share/hashcat/rules/best64.rule
# Result: ProjectNova! -> matches the hash
```

hashcat: Essential Modes for CTF

Mode (-m)	Hash Type	Example
0	MD5	5d41402abc4b2a76b9719d911017c592
100	SHA-1	aaf4c61ddcc5e8a2...
1400	SHA-256	2cf24dba5fb0a30e...
1800	sha512crypt	\$6\$rounds=...
3200	bcrypt	\$2a\$12\$...
1000	NTLM	a4f49c406510bdca...
13100	Kerberoast	\$krb5tgs\$23\$...

```
# Attack modes
hashcat -a 0 hash.txt wordlist.txt           # dictionary
hashcat -a 0 hash.txt wordlist.txt -r best64 # dictionary + rules
```

```
hashcat -a 3 hash.txt ?a?a?a?a?a?a      # brute-force (6 chars)
hashcat -a 6 hash.txt wordlist.txt ?d?d?d?d  # hybrid (word + 4 digits)
```

Steganography: Hidden Data in Files

Steganography hides data within other files — images, audio, PDFs, even text. Unlike encryption, the goal is to hide the *existence* of the message.

Common Hiding Techniques

- **LSB embedding** — data in least significant bits of image pixels
- **Appended data** — extra bytes after the file's EOF marker
- **Metadata** — EXIF, comments, XMP fields
- **Whitespace** — trailing spaces / tabs encode binary

Detection Clues

- File size larger than expected for the content
- Statistical anomalies in pixel/byte distribution
- Extra data after EOF markers (JPEG: `FF D9`)
- Unusual metadata fields

Steganography: Tools & Extraction

```
# Step 1: Check for appended data
$ binwalk challenge.png

DECIMAL      HEXADECIMAL    DESCRIPTION
0            0x0            PNG image, 800 x 600, 8-bit/color RGBA
87           0x57           Zlib compressed data
94231        0x17017        Zip archive data, name: secret.txt
94389        0x170B5        End of Zip archive

# Step 2: Extract embedded files
$ binwalk -e challenge.png
$ cat _challenge.png.extracted/secret.txt
flag{binwalk_finds_everything}

# Step 3: Check metadata
$ exiftool challenge.png | grep -i "comment\\|flag\\|secret"
```

AI-Assisted Steganography Detection

AI can analyze hex dumps, file metadata, and statistical properties to identify hidden data.

```
import anthropic

# Feed file analysis results to AI
analysis_data = """
File: mystery.png (245 KB - unusually large for 100x100 image)

Hex dump of last 128 bytes:
00 00 00 00 49 45 4e 44 ae 42 60 82 50 4b 03 04
14 00 00 00 08 00 b4 7a 55 59 8a 2c bf 3e 2d 00
00 00 2b 00 00 00 0a 00 1c 00 73 65 63 72 65 74

Strings found after PNG IEND marker:
PK\x03\x04 (ZIP magic bytes)
secret.txt
```

The AI-Assisted Analysis Workflow

1. **Triage** — `file`, `strings`, `binwalk`, `exiftool` to identify what you have
2. **Disassemble** — Ghidra or radare2 to decompile interesting functions
3. **AI Analysis** — Feed decompiled code to Claude: "What does this do? How to exploit it?"
4. **Identify Crypto** — Recognize algorithm patterns, key management flaws
5. **AI Attack Plan** — "Given this cipher with a 4-byte repeating key, write a known-plaintext attack"
6. **Extract & Decode** — CyberChef for multi-layer decoding, hashcat for hashes
7. **Verify** — Confirm the flag format and submit

Key Principle

AI does not replace your understanding — it **accelerates** it. You still need to know what questions to ask and how to validate the answers.

CTF Challenge: CryptoVault

Mission: Reverse engineer the binary and break the custom encryption to recover the flag.

Scenario

You have intercepted `cryptovault`, an ELF binary that encrypts secrets. A user's encrypted vault file `vault.enc` was also recovered. Your goal: decrypt the vault and extract the flag.

```
# Initial recon
$ file cryptovault
cryptovault: ELF 64-bit LSB executable, x86-64, not stripped

$ strings cryptovault | grep -i interesting
CryptoVault v1.0 - Unbreakable Encryption(tm)
Enter master password:
Encrypting with military-grade algorithm...
ENCRYPTED_HEADER_V1

$ strings vault.enc | head -2
```

ENCRYPTED_HEADER_V1

CTF Challenge: Reversing the Binary

```
// Ghidra decompilation of encrypt_vault()
void encrypt_vault(char *data, int len, char *password) {
    char key[32];
    int i;

    // "Key derivation" - just MD5 of the password!
    md5_hash(password, key);

    // "Military-grade encryption" - repeating XOR with MD5 hash
    for (i = 0; i < len; i++) {
        data[i] = data[i] ^ key[i % 16];
    }
}

// Ghidra also reveals the password check:
```

Vulnerabilities Found

- **MD5 for key derivation** — crackable hash
- **Repeating XOR** — vulnerable to known-plaintext attack

- **Hardcoded hash comparison** — crack MD5 to get password
-

CTF Challenge: Breaking the Encryption

```
import hashlib

# Step 1: Crack the MD5 hash
# 5f4dcc3b5aa765d61d8327deb882cf99 is MD5 of "password"
# (found instantly via hashcat or online lookup)
master_password = "password"

# Step 2: Derive the XOR key (same as the binary does)
key = hashlib.md5(master_password.encode()).digest()
print(f"XOR key: {key.hex()}")
# 5f4dcc3b5aa765d61d8327deb882cf99

# Step 3: Decrypt the vault
with open("vault.enc", "rb") as f:
    header = f.readline() # skip ENCRYPTED HEADER V1
```

Notes: This challenge combines reversing (understanding the encrypt function), crypto (cracking MD5 + breaking XOR), and the AI workflow (using Claude to analyze the decompiled code).

Common Pitfalls & Tips

Pitfalls

- Jumping into Ghidra before running `strings`
- Ignoring the binary's linked libraries as clues
- Assuming crypto is strong without checking implementation
- Overlooking base64 / hex encoding layers
- Forgetting to check file metadata for stego

Tips

- Always start with `file`, `strings`, `ltrace`
- Use `ltrace` / `strace` to see library calls at runtime
- XOR with a known plaintext prefix recovers key bytes
- Check CrackStation / online databases before brute-forcing
- Ask AI to explain unfamiliar assembly patterns

Key Takeaways

1. **Triage first** — `file`, `strings`, `binwalk` before disassembly
2. **AI accelerates reversing** — paste decompiled code and ask for analysis
3. **Know the vulnerability classes** — buffer overflow, format string, use-after-free
4. **Weak crypto is everywhere** — MD5, XOR, ECB mode, and poor key management
5. **Known-plaintext attacks** are powerful against XOR and stream ciphers
6. **Steganography** — check file size, metadata, appended data, and pixel statistics
7. **CyberChef + hashcat + AI** form the core crypto analysis toolkit

Next Lesson

Lesson 07: Configuration & Infrastructure Auditing — Finding misconfigurations, hardcoded secrets, and infrastructure weaknesses with AI assistance.

