

Binary & Cryptographic Analysis

Reverse engineering compiled programs, breaking weak cryptography, and leveraging AI to accelerate analysis in CTF challenges and security assessments.

WARNING: Legal & Ethical Boundaries. Only analyze binaries and cryptographic systems that you have explicit legal authorization to examine. This includes CTF challenge files, software you own, and systems covered by a signed scope-of-work agreement. Unauthorized reverse engineering may violate the Computer Fraud and Abuse Act (CFAA), the DMCA, or equivalent laws in your jurisdiction. When in doubt, do not proceed.

Why Binary Analysis Matters

Software does not always arrive as readable source code. Malware samples, proprietary firmware, embedded device images, and CTF challenges all present themselves as compiled binaries. The ability to examine these artifacts — understanding what they do, how they do it, and where they are vulnerable — is a core skill for security practitioners.

In CTF competitions, binary challenges (often categorized as "reversing" or "pwn") consistently appear at every difficulty level. Competitors must extract flags hidden inside executables, exploit memory corruption vulnerabilities, or defeat custom obfuscation schemes. In professional security work, binary analysis underpins malware reverse engineering, vulnerability research, and firmware auditing. Whether you are competing or consulting, the methodology is the same: systematic examination of structure, logic, and data flow.

ELF and PE File Formats

The two dominant executable formats you will encounter are **ELF** (Executable and Linkable Format) on Linux/Unix systems and **PE** (Portable Executable) on Windows.

ELF Format

ELF files begin with a fixed header containing a magic number (`7f 45 4c 46` , or `\x7fELF`), followed by metadata describing the target architecture, entry point address, and section/program header tables. Key sections include:

- **.text** — executable machine code
- **.data** — initialized global variables
- **.bss** — uninitialized global variables
- **.rodata** — read-only data (string literals, constants)
- **.plt / .got** — procedure linkage table and global offset table for dynamic linking
- **.symtab / .strtab** — symbol and string tables (stripped in release builds)

PE Format

PE files start with an MS-DOS stub (magic bytes `4d 5a` , or "MZ"), followed by the PE signature, COFF header, and optional header. Sections typically include `.text` , `.rdata` , `.data` , and `.rsrc` (resources). The Import Address Table (IAT) lists functions the binary calls from external DLLs — examining the IAT often reveals the binary's capabilities at a glance (networking, file I/O, cryptographic operations).

The Analysis Toolchain

Effective binary analysis requires layered tooling, starting with quick triage and progressing to deep disassembly.

Quick triage: file, strings, xxd

```
# Identify the file type, architecture, and linking
file challenge_binary
# challenge_binary: ELF 64-bit LSB executable, x86-64, dynamically linked

# Extract printable strings (minimum length 6)
strings -n 6 challenge_binary | head -30

# Look for specific patterns: flags, URLs, keys
strings challenge_binary | grep -iE 'flag|ctf|key|password|secret'
```

```
# Hex dump the first 128 bytes to inspect headers
xxd -l 128 challenge_binary
```

The `strings` command alone solves a surprising number of beginner CTF challenges where flags are embedded as plaintext literals.

Structural analysis: objdump, readelf

```
# Disassemble the .text section
objdump -d -M intel challenge_binary | less

# Show section headers and their addresses
readelf -S challenge_binary

# List dynamic symbols (imported functions)
readelf --dyn-syms challenge_binary

# Display program headers (memory layout at runtime)
readelf -l challenge_binary
```

Deep analysis: Ghidra and radare2

Ghidra is an open-source reverse engineering framework developed by the NSA. Its decompiler converts assembly back into C-like pseudocode, dramatically accelerating analysis. Import a binary, let auto-analysis complete, then navigate functions in the decompiler view. Ghidra excels at renaming variables, adding comments, and cross-referencing function calls.

radare2 is a command-line reverse engineering toolkit favored for scripting and automation. Its key commands include `aaa` (analyze all), `afl` (list functions), `pdf @main` (print disassembly of main), and `vv` (visual graph mode). While the learning curve is steep, radare2's scriptability via `r2pipe` makes it invaluable for automated analysis pipelines.

Static vs Dynamic Analysis

Static analysis examines the binary without executing it. You read disassembly, trace control flow graphs, and identify patterns in the code. It is safe (no malicious code runs) and gives a complete view of the program, but can be slow and is easily defeated by obfuscation or packing.

Dynamic analysis runs the binary (typically in a sandbox or under a debugger) and observes its behavior. Tools like `gdb`, `strace`, `ltrace`, and `valgrind` reveal runtime behavior: system calls, memory allocations, network connections, and file operations. Dynamic analysis cuts through obfuscation but only shows the code paths actually executed.

The most effective approach combines both: use static analysis to build a mental model of the program, then confirm hypotheses dynamically. In CTF contexts, setting breakpoints at key comparison functions (like `strcmp` or custom check routines) and inspecting register/memory contents is often the fastest path to a flag.

Common Binary CTF Challenges

Reverse engineering

The binary takes user input and checks it against an expected value. Your task is to determine the correct input (the flag) by understanding the check logic. Approaches include: reading the decompiled code to reconstruct the algorithm, using a constraint solver (like Z3) to find valid inputs, or patching the binary to bypass checks.

Finding hidden strings

Flags may be XOR-encoded, base64-encoded, split across multiple variables, or constructed at runtime from character arrays. Look for loops that build strings character by character, XOR operations with a constant key, or calls to decoding functions.

Buffer overflows

The binary reads input into a fixed-size buffer without bounds checking. By overflowing the buffer, you overwrite the return address on the stack to redirect execution.

Common targets include jumping to a "win" function that prints the flag, or injecting shellcode. Modern mitigations (ASLR, stack canaries, NX) add complexity but can often be bypassed in CTF settings.

Cryptographic Analysis

Cryptographic challenges in CTFs range from textbook cipher breaks to exploiting subtle implementation flaws in real protocols. The first step is always identification: what algorithm or scheme is in use?

Identifying encryption algorithms

Look for telltale constants in the binary or source code. AES uses the S-box constant `0x63` and the round constant table. DES uses specific permutation tables. RSA involves large integer operations and the relationship $n = p * q$. In Python-based challenges, import statements (`from Crypto.Cipher import AES`) are direct indicators.

Weak crypto patterns

Common weaknesses exploited in CTFs include:

- **ECB mode** — identical plaintext blocks produce identical ciphertext blocks, leaking patterns
- **Reused nonces/IVs** — in stream ciphers or CTR mode, nonce reuse allows XOR of plaintexts
- **Small RSA exponents** — `e = 3` with no padding allows cube-root attacks
- **Textbook RSA** — no padding scheme enables chosen-ciphertext attacks
- **Custom "encryption"** — XOR with a repeating key, Caesar/substitution ciphers, or home-grown algorithms

Classic Crypto Attacks

Frequency analysis

Against substitution ciphers, letter frequency distributions in the ciphertext reveal the mapping. English text has characteristic frequencies (E ~12.7%, T ~9.1%, A ~8.2%). Bigram and trigram analysis further constrains possibilities. This technique breaks monoalphabetic substitution and Vigenere ciphers (once the key length is determined via Kasiski examination or index of coincidence).

Known-plaintext attacks

If you know (or can guess) part of the plaintext, you can recover XOR keys directly. For a repeating-key XOR cipher: `key = known_plaintext XOR ciphertext`. In CTFs, flag

formats like `CTF{` or `flag{` often provide the known plaintext needed to bootstrap the attack.

Padding oracle attacks

When a server reveals whether CBC-mode decryption produced valid padding, an attacker can decrypt any ciphertext byte by byte. The attack sends modified ciphertext blocks and observes the server's response (valid padding vs. padding error). Tools like `PadBuster` automate this process. The fix is to use authenticated encryption (AES-GCM) rather than CBC with separate MAC checks.

AI-Assisted Binary Analysis

Large language models like Claude can significantly accelerate binary and cryptographic analysis in several ways:

- **Explaining disassembly.** Paste a function's assembly or decompiled pseudocode into Claude and ask it to explain the logic, identify the algorithm, and suggest what the function does. Claude can recognize common patterns like sorting algorithms, hash functions, and string manipulation routines.
- **Identifying crypto algorithms.** Share constants, S-box values, or pseudocode with Claude to identify whether the code implements AES, DES, RC4, RSA, or a custom scheme. Claude can also spot implementation errors that create vulnerabilities.
- **Suggesting attack vectors.** Describe the binary's behavior and protections, and Claude can suggest exploitation strategies: ROP chain construction approaches, format string exploitation techniques, or crypto attack methodologies appropriate for the identified weakness.
- **Writing exploit scripts.** Claude can generate pwntools scripts, Z3 constraint solver setups, or crypto attack implementations based on your analysis findings.

Tip: When sharing disassembly with Claude, include the function signature, calling convention, and any context about what the binary does overall. More context produces more accurate analysis.

Python Tools for Binary & Crypto Work

pwntools

`pwntools` is the standard Python library for exploit development and CTF binary challenges. It handles process interaction, networking, packing/unpacking, shellcode generation, and ROP chain construction.

```
from pwn import *

# Connect to a remote challenge
io = remote('challenge.ctf.example', 1337)

# Or run a local binary
io = process('./challenge_binary')

# Send a payload with a packed address
payload = b'A' * 64 # padding to reach return address
payload += p64(0x00401234) # overwrite return address (little-endian)
io.sendline(payload)

# Receive output until we find the flag
response = io.recvuntil(b'flag{')
flag = b'flag{' + io.recvuntil(b'}')
print(flag.decode())
```

struct module

Python's built-in `struct` module packs and unpacks binary data according to format strings. Essential for parsing file headers, network protocols, and binary structures.

```
import struct

# Read and parse an ELF header
with open('challenge_binary', 'rb') as f:
    header = f.read(64)

# Unpack fields: magic (4 bytes), class, endian, version, OS/ABI
magic = header[:4]
print(f"Magic: {magic}") # b'\x7fELF'

# Unpack the entry point address (offset 24, 8 bytes for 64-bit ELF)
entry_point = struct.unpack_from('<Q', header, 24)[0]
print(f"Entry point: 0x{entry_point:016x}")
```

```
# Pack an address in little-endian 64-bit format
addr = struct.pack('<Q', 0x00401234)
print(f"Packed: {addr.hex()}") # 3412400000000000
```

Crypto analysis with hashlib and custom scripts

```
import hashlib
from itertools import product
from string import ascii_lowercase

# Brute-force a short hash prefix (proof-of-work challenge)
target_prefix = "00000"
for length in range(1, 8):
    for combo in product(ascii_lowercase, repeat=length):
        candidate = ''.join(combo)
        digest = hashlib.sha256(candidate.encode()).hexdigest()
        if digest.startswith(target_prefix):
            print(f"Found: '{candidate}' -> {digest}")
            break

# XOR decryption with a known key
def xor_decrypt(ciphertext: bytes, key: bytes) -> bytes:
    """Decrypt repeating-key XOR."""
    return bytes(c ^ key[i % len(key)] for i, c in enumerate(ciphertext))

ciphertext = bytes.fromhex('1c0011001a00410d00')
# Try recovering key using known plaintext "CTF{"
known = b'CTF{'
recovered_key = xor_decrypt(ciphertext[:len(known)], known)
print(f"Recovered key bytes: {recovered_key.hex()}")

# Decrypt full message with recovered key
```

Putting It All Together

A systematic approach to binary and crypto challenges follows this workflow:

1. **Triage.** Run `file` and `strings`. Check for obvious flags, known file signatures, and embedded metadata.
2. **Static analysis.** Load into Ghidra or radare2. Identify `main()`, trace the control flow, and locate key functions (input handling, validation, encryption/decryption).

3. **AI consultation.** Paste complex functions into Claude for explanation and algorithm identification. Ask for attack strategy suggestions.
4. **Dynamic analysis.** Run under `gdb` with breakpoints at critical comparisons or crypto operations. Inspect memory contents and register values.
5. **Exploit development.** Write a pwntools script or crypto attack implementation. Test locally, then run against the remote target.
6. **Verify.** Confirm the flag format and submit.

Further Reading

- [Ghidra](#) — NSA's open-source reverse engineering framework
- [radare2](#) — command-line reverse engineering toolkit
- [pwntools documentation](#) — Python CTF and exploit development library
- [Cryptopals Challenges](#) — hands-on cryptographic attack exercises
- [Reverse Engineering for Beginners](#) by Dennis Yurichev — comprehensive free textbook
- [LiveOverflow](#) — video walkthroughs of binary exploitation and CTF challenges

Download Files

These files contain synthetic binary and crypto challenge data. All files are plain text (.txt) and safe to open.

↓ [binary-hexdump.txt](#) ~5 KB

↓ [strings-output.txt](#) ~4 KB

↓ [disassembly-snippet.txt](#) ~11 KB

↓ [crypto-challenge.txt](#) ~12 KB