

Lab Guide

Four exercises: strings triage, hex-dump interpretation, disassembly reading, and cryptographic puzzle solving — all with AI as your reverse-engineering assistant.

Authorized Use Only

All binaries, dumps, and crypto challenges are **synthetic**. They are designed to teach pattern recognition, not to model real malware. Reverse-engineer only software you are authorized to analyze.

Prerequisites

- Completed the [Theory](#) reading
- Access to an AI assistant (Claude, ChatGPT, or local LLM)
- Downloaded all four files into a folder named `binary-lab/`
- Familiarity with: x86-64 assembly basics, C calling conventions, ASCII/hex conversion, basic crypto (XOR, AES, RSA)
- Optional: `radare2`, `Ghidra`, or `xxd`, `CyberChef` for cross-checking

Lab Scenario

AcmeCorp's incident responders pulled a suspicious binary off a workstation. They have extracted strings, dumped hex, captured a disassembly snippet, and found an encrypted blob with hints suggesting it contains flag data. Your job: reconstruct what the binary does, extract any secrets, and decrypt the blob — all using only text artifacts and an AI assistant.

Download Lab Files

Save all four files into a folder named `binary-lab/` before starting the exercises.

↓ [strings-output.txt](#) ~4 KB

↓ [binary-hexdump.txt](#) ~5 KB

↓ [disassembly-snippet.txt](#) ~11 KB

↓ [crypto-challenge.txt](#) ~12 KB

Exercise 1 — Strings Triage

Open `strings-output.txt`. The output of `strings -n 6` is the cheapest source of high-value intel from a binary.

Task 1.1 — Extract everything actionable

Below is the output of ``strings`` on an unknown binary. Extract:

1. Any URLs, IP addresses, or hostnames (C2 indicators).
2. Any file paths (Windows/Linux) — note OS hints.
3. Any registry keys, service names, or scheduled task names (persistence indicators).
4. Any command-line strings (cmd.exe, powershell, bash commands being built).
5. Any format strings like "%s" "%d" — hint at printf/sprintf usage and sometimes reveal log message templates.
6. Any obviously base64/hex-encoded blobs (decode them).
7. Any library/function names imported (DLL names, syscalls).
8. Any version numbers, compiler banners, or build paths (attribution hints).

Then infer:

- What language/compiler produced this binary?
- What platform is it for?
- What does it probably DO?
- What is the most actionable IOC for a defender?

```
[paste strings-output.txt]
```

✓ **Self-check:** Strings output usually contains at least one high-value string (URL, path, or command). You should be able to make a confident guess about the binary's purpose within 5 minutes.

Exercise 2 — Hex Dump Interpretation

Open `binary-hexdump.txt`. A hex dump is structured data disguised as random numbers.

Task 2.1 — Identify the file format and extract structure

Below is a hex dump. Analyze it as a binary forensics analyst:

1. Identify the file format from the magic bytes at offset 0 (ELF, PE, ZIP, PNG, JPEG, PDF, Mach-O, etc.).
2. Parse the header fields:
 - For PE: DOS stub, PE signature, machine type, sections
 - For ELF: class, endianness, OS ABI, type, machine
 - For ZIP/archive: local file headers, filenames
 - For PNG/JPEG: dimensions, color mode
3. Highlight any anomalies:
 - Non-standard section names
 - Unusual sizes or offsets
 - Embedded file signatures (nested formats)
 - High-entropy regions (likely encryption / packing)
 - Hardcoded strings in unexpected places
4. Decode any ASCII text hiding in the hex.
5. Calculate and flag: overlay data, padding regions, overlapping sections, size mismatches.

```
[paste binary-hexdump.txt]
```

✓ **Self-check:** You should be able to state the file format with confidence and identify at least one interesting structural element (section, embedded file, or anomaly).

Exercise 3 — Disassembly Reading

Open `disassembly-snippet.txt`. This is x86-64 assembly from a single function.

Task 3.1 — Reverse the function to pseudo-C

Below is x86-64 disassembly of a single function. Do the following:

1. Identify the calling convention (System V AMD64 vs. Microsoft x64).
2. Count the local variables by stack frame size and usage.
3. Identify loops, branches, and function calls.
4. Translate the assembly into pseudo-C that preserves behavior.
Use descriptive names where you can infer purpose.
5. Identify any cryptographic primitives by constants or structure:
 - SHA-256 round constants
 - AES S-box
 - RC4 KSA pattern (256-byte array init)
 - XOR key loops
 - RSA / bignum operations
6. Identify any syscalls or library calls and what they do.
7. Determine the function's purpose in one sentence.
8. Flag any suspicious behavior (anti-debug, anti-VM, self-modification, process hollowing, code injection).

[paste disassembly-snippet.txt]

✓ **Self-check:** Your pseudo-C should be short and readable. If the AI gives you something that is just "disassembly with comments," ask it to "rewrite as the smallest possible C function that matches this behavior."

Exercise 4 — Cryptographic Challenge

Open `crypto-challenge.txt`. This file contains ciphertexts, keys, hashes, or other cryptographic artifacts with hints about how they were generated.

Task 4.1 — Identify the primitive and break it

Below is a cryptographic challenge. For each artifact in the file:

1. Identify the primitive being used:
 - Symmetric: AES (mode?), DES, RC4, XOR, ChaCha
 - Asymmetric: RSA (key size?), DSA, ECC
 - Hash: MD5, SHA-1, SHA-256, bcrypt, argon2
 - Encoding: base64, hex, ROT13
2. Identify any weaknesses:
 - Reused keys/nonces (ECB-mode, stream cipher reuse)
 - Small RSA modulus or exponent
 - Weak passwords + fast hash (crackable)
 - Known plaintext available
 - Short/predictable IVs
3. Write working Python (or CyberChef recipe) to:
 - Decrypt the ciphertext
 - Crack the hash
 - Factor the modulus
 - Recover the key
4. Output the plaintext (flag) and the recovery technique.
5. If multiple artifacts exist, solve them in order from easiest to hardest.

[paste crypto-challenge.txt]

✓ **Self-check: You should recover at least one plaintext and understand WHY the cryptography failed. "I got the flag" is not enough — you must be able to explain the attack in one sentence.**

Final Exercise — Binary Behavior Reconstruction

Task 5.1 — Cross-artifact synthesis

Here are four analyses of four artifacts extracted from the same unknown binary. Reconstruct:

1. What the binary is (malware family? legitimate tool? custom?).
2. What it does on execution (persistence, C2, data theft, destruction, lateral movement).
3. The C2 protocol and endpoints (from strings + crypto analysis).

4. Any keys, passwords, or secrets hardcoded in the binary.
5. The plaintext of any encrypted data it was carrying.

Then produce:

- A YARA rule that would detect this binary by unique strings and byte patterns.
- A Sigma rule that would detect its behavior on an endpoint.
- A one-paragraph threat-intel summary for a SOC.

Format as markdown.

✓ **Self-check: The final report should connect all four artifacts — e.g., "the strings reveal a URL, the hex shows an embedded encrypted config, the disassembly shows an AES decrypt routine, and the crypto challenge provides the key that decrypts the config to reveal C2 infrastructure."**

Reflection

- Which artifact gave the most information per byte? Which took the longest to interpret?
- How well did the AI handle raw assembly? Did it hallucinate functions that were not there?
- What was the hardest cryptographic primitive for the AI to identify without explicit hints?
- In a real incident, which of these four artifacts would you pull first if you only had time for one?

What's Next

Lesson 07 moves from code and binaries to **configuration and infrastructure** — nmap scans, container specs, IAM policies, and nginx configs.