

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.