Assignment 1.2: Comprehensive Reconnaissance Methodology

WAS600 – Web Security

Adeyemi Folarin

February 2026

Abstract

This document presents a structured, phased methodology for conducting web application reconnaissance during authorized bug bounty engagements. The methodology spans passive intelligence gathering through active enumeration and concludes with automation strategies and reporting workflows. Each phase includes specific tooling, concrete command-line examples with representative output, and guidance on interpreting results. The goal is to provide a repeatable, thorough recon framework that maximizes target surface discovery while remaining within program scope.

Contents

1 Introduction **4**

2 Phase 1: Passive Reconnaissance **4**
2.1 WHOIS and Reverse WHOIS . . . 4
2.2 DNS Enumeration . . . 5
2.3 Certificate Transparency Logs . . . 6
2.4 Google Dorking . . . 7
2.5 Shodan and Censys Reconnaissance . . . 8
2.6 OSINT: Social Media and Technology Inference . . . 8
2.7 Wayback Machine Reconnaissance . . . 9
2.8 GitHub Reconnaissance . . . 10

3 Phase 2: Active Recon – Subdomain Discovery **10**
3.1 Subdomain Enumeration Tools . . . 11
3.2 Subdomain Validation with httpx . . . 12
3.3 Resolving and Filtering Live Hosts . . . 13
3.4 Virtual Host Discovery . . . 13

4 Phase 3: Active Recon – Service & Content Discovery **13**
4.1 Port Scanning . . . 13
4.2 Directory and Content Brute-Forcing . . . 14
4.3 Technology Fingerprinting . . . 16
4.4 JavaScript File Analysis . . . 16
4.5 Parameter Discovery . . . 17

5 Phase 4: API Reconnaissance **17**
5.1 API Endpoint Discovery with Burp Suite . . . 17
5.2 API Testing with Postman . . . 18
5.3 API Fuzzing with Kiterunner . . . 18
5.4 Swagger and OpenAPI Specification Hunting . . . 19

6 Phase 5: Vulnerability Scanning **19**
6.1 Nuclei . . . 19
6.2 Nikto . . . 21
6.3 Burp Suite Scanner . . . 21

7 Phase 6: Automation and Scripting **21**
7.1 Bash Automation: Full Recon Pipeline . . . 22
7.2 Python Automation: Subdomain + httpx + Nuclei Chain . . . 23
7.3 TomNomNom Tools Integration . . . 26
7.4 Cristi Vlad Scripts Reference . . . 26

8 Phase 7: Reporting and Documentation **27**
8.1 Note-Taking with Obsidian . . . 27
8.2 Organizing Findings . . . 28
8.3 Report Structure . . . 28

9 Workflow Diagram **28**

10 Tool Reference Table **29**

11 Practical Demonstration: Juice Shop & Postman API Recon **31**
11.1 Target Discovery and Reconnaissance 31
11.1.1 Application Overview 31
11.1.2 API Endpoint Discovery via Browser DevTools 31
11.1.3 Examining API Responses 32
11.2 Vulnerability Discovery 32
11.2.1 Hidden Content: Score Board 32
11.2.2 SQL Injection on Login 33
11.2.3 Accessing the Administration Panel 33
11.3 API Reconnaissance with Postman 33
11.3.1 Environment Configuration 34
11.3.2 Collection Setup 34
11.3.3 Product Enumeration 35
11.3.4 Search Functionality Testing 35
11.3.5 Authentication: Login Flow 35
11.3.6 Authenticated Requests 36
11.3.7 User Registration Endpoint 36
11.3.8 Complete API Collection 36
11.4 Key Takeaways from Practical Exercise 37

12 Conclusion **38**

1 Introduction

Reconnaissance is the foundational phase of any security assessment. The quality and completeness of recon directly determines the attack surface available for subsequent testing. A thorough recon phase uncovers subdomains, services, API endpoints, technology stacks, and potential misconfigurations that narrower approaches would miss.

This methodology is organized into seven sequential phases:

1. **Passive Reconnaissance** – Gathering information without directly interacting with the target's infrastructure.
2. **Subdomain Discovery** – Actively enumerating subdomains to expand the attack surface.
3. **Service & Content Discovery** – Identifying running services, open ports, and hidden content.
4. **API Reconnaissance** – Discovering and mapping API endpoints.
5. **Vulnerability Scanning** – Automated scanning for known vulnerability patterns.
6. **Automation & Scripting** – Chaining tools into repeatable pipelines.
7. **Reporting & Documentation** – Organizing and presenting findings.

Each phase builds on the outputs of the previous one. Passive recon feeds domain lists into subdomain discovery, which feeds live hosts into service enumeration, and so on. This pipeline approach ensures comprehensive coverage and efficient use of testing time.

Scope compliance is enforced at every phase. Before any active testing, the target program's scope document must be reviewed, and all discovered assets must be validated against the in-scope list. Out-of-scope assets are flagged and excluded from further testing.

2 Phase 1: Passive Reconnaissance

Passive reconnaissance collects information about the target without sending any traffic to the target's infrastructure. This phase is generally safe from a legal perspective and produces no logs on the target's systems.

2.1 WHOIS and Reverse WHOIS

WHOIS records reveal domain registration details including registrant information, name servers, registration dates, and associated organizations.

Listing 1: Basic WHOIS lookup

```
# Standard WHOIS query
whois github.com
```

Listing 2: WHOIS output (truncated)

```
Domain Name: GITHUB.COM
Registry Domain ID: 1264983250_DOMAIN_COM-VRSN
Registrar: MarkMonitor Inc.
Creation Date: 2007-10-09T18:20:50Z
Updated Date: 2024-09-07T09:28:09Z
Name Server: DNS1.P08.NSONE.NET
Name Server: DNS2.P08.NSONE.NET
Name Server: DNS3.P08.NSONE.NET
Name Server: DNS4.P08.NSONE.NET
Name Server: NS-1707.AWSDNS-21.CO.UK
Name Server: NS-520.AWSDNS-01.NET
```

Key observations from WHOIS data include name server providers (indicating hosting infrastructure), registrar choice, and registration history. MarkMonitor is a brand protection registrar commonly used by large enterprises.

Reverse WHOIS identifies other domains registered by the same organization:

Listing 3: Reverse WHOIS via ARIN

```
# ARIN organization lookup for IP range ownership
whois -h whois.arin.net "o GitHub, Inc."

# Using amass for reverse WHOIS
amass intel -org "GitHub, Inc."
```

This can reveal subsidiary domains, development environments, and staging servers that may be in scope.

2.2 DNS Enumeration

DNS records map the target's infrastructure. Key record types include A (IPv4), AAAA (IPv6), MX (mail), NS (name servers), TXT (SPF, DKIM, verification records), CNAME (aliases), and SOA (zone authority).

Listing 4: Comprehensive DNS enumeration

```
# Query all record types
dig github.com ANY +noall +answer

# Specific record types
dig github.com A +short
dig github.com MX +short
dig github.com TXT +short
dig github.com NS +short

# Zone transfer attempt (usually blocked)
dig axfr github.com @ns-520.awsdns-01.net

# Reverse DNS for discovered IPs
dig -x 140.82.121.3 +short
```

Listing 5: DNS query output examples

```
$ dig github.com A +short
140.82.121.3

$ dig github.com MX +short
1 aspmx.l.google.com.
5 alt1.aspmx.l.google.com.
5 alt2.aspmx.l.google.com.
10 alt3.aspmx.l.google.com.

$ dig github.com TXT +short
"v=spf1 ip4:192.30.252.0/22 include:_netblocks.google.com ..."
"MS=ms58704441"
"docusign=..."
```

TXT records are particularly valuable — they can reveal third-party service integrations (Google Workspace, DocuSign, Microsoft 365), which expand the target's technology profile.

The `host` command provides quick lookups:

Listing 6: Quick DNS with host

```
host -a github.com
host -t ns github.com
```

2.3 Certificate Transparency Logs

Certificate Transparency (CT) is a public logging framework for TLS certificates. Every certificate issued by a trusted CA is logged, making CT logs a rich source for subdomain discovery.

Listing 7: CT log queries

```
# Using crt.sh (most common)
curl -s "https://crt.sh/?q=%25.github.com&output=json" | \
    jq -r '.[].name_value' | sort -u

# Using certspotter API
curl -s "https://api.certspotter.com/v1/issuances?domain=github.com\
&include_subdomains=true&expand=dns_names" | \
    jq -r '.[].dns_names[]' | sort -u
```

Listing 8: CT log results (sample)

```
github.com
*.github.com
*.github.io
assets-cdn.github.com
avatars.githubusercontent.com
codeload.github.com
```

```
gist.github.com
pages.github.com
raw.githubusercontent.com
```

CT logs frequently reveal internal subdomains, staging environments, and deprecated services that may still be accessible. Wildcard entries (`*.github.com`) indicate the target uses wildcard certificates, meaning additional subdomains likely exist beyond what CT logs show.

2.4 Google Dorking

Google dorking uses advanced search operators to find indexed content that reveals target infrastructure, configuration files, error messages, and other sensitive information.

Listing 9: Google dork examples

```
# Find subdomains indexed by Google
site:github.com -www

# Find specific file types
site:github.com filetype:pdf
site:github.com filetype:xml
site:github.com filetype:conf OR filetype:cfg OR filetype:env

# Find login pages
site:github.com inurl:login OR inurl:signin OR inurl:auth

# Find exposed directories
site:github.com intitle:"index of"

# Find error messages revealing stack traces
site:github.com intext:"stack trace" OR intext:"error occurred"

# Find API documentation
site:github.com inurl:api inurl:docs OR inurl:swagger OR inurl:graphql

# Find admin panels
site:github.com inurl:admin OR inurl:dashboard OR inurl:panel

# Combine operators for precision
site:*.github.com -www -docs inurl:api filetype:json
```

Key operators reference:

- `site:` – Restrict results to a domain
- `inurl:` – Match text in URL path
- `intitle:` – Match text in page title
- `intext:` – Match text in page body
- `filetype:` – Restrict to file extension
- `ext:` – Alias for filetype
- `cache:` – View cached version

- `-` (minus) – Exclude term
- `OR` / `|` – Boolean OR
- `""` (quotes) – Exact match

Automated dorking tools like `dorkscanner` and `pagodo` can run batches of dorks, but manual review of results is essential to avoid false positives.

2.5 Shodan and Censys Reconnaissance

Shodan and Censys are internet-wide scanning databases that index open ports, services, banners, SSL certificates, and HTTP responses across the entire IPv4 address space.

Listing 10: Shodan CLI queries

```
# Search by organization
shodan search "org:GitHub" --fields ip_str,port,product

# Search by hostname
shodan search "hostname:github.com" --fields ip_str,port,org

# Search by SSL certificate
shodan search "ssl.cert.subject.CN:github.com"

# Get host details
shodan host 140.82.121.3

# Count results for a query
shodan count "org:GitHub port:443"
```

Listing 11: Censys CLI queries

```
# Search hosts by domain
censys search "services.tls.certificates.leaf.names: github.com"

# Search by autonomous system
censys search "autonomous_system.name: GitHub"
```

These databases reveal services that may not be discoverable through DNS alone, including non-standard ports, legacy services, and misconfigured systems. Cross-referencing Shodan results with the target's known IP ranges identifies internet-facing assets.

2.6 OSINT: Social Media and Technology Inference

Open-source intelligence beyond technical records includes:

- **LinkedIn job postings** – Technology requirements in job descriptions reveal the internal tech stack (e.g., "Experience with Ruby on Rails, Kubernetes, Elasticsearch" confirms those technologies are in use).
- **GitHub repositories** – The target's own public repositories may contain configuration files, deployment scripts, and infrastructure-as-code that reveals internal architecture.

- **Conference talks and blog posts** – Engineering blog posts often detail architecture decisions, tool choices, and infrastructure details.
- **Pastebin and paste sites** – Historical data leaks may reference internal hostnames, endpoints, or credentials.
- **BuiltWith / Wappalyzer (passive mode)** – Technology profiling based on HTTP headers, JavaScript libraries, and other fingerprints.

Listing 12: Technology inference commands

```
# Check HTTP headers for technology hints
curl -sI https://github.com | grep -iE "server|x-powered|x-frame|x-content"

# Check robots.txt for hidden paths
curl -s https://github.com/robots.txt

# Check security.txt
curl -s https://github.com/.well-known/security.txt

# Check sitemap
curl -s https://github.com/sitemap.xml | head -50
```

2.7 Wayback Machine Reconnaissance

The Wayback Machine archives historical snapshots of websites. Historical content can reveal removed pages, old API endpoints, deprecated functionality, and previously exposed sensitive files.

Listing 13: Wayback Machine tools

```
# Install waybackurls (Go tool by tomnomnom)
go install github.com/tomnomnom/waybackurls@latest

# Fetch all archived URLs for a domain
echo "github.com" | waybackurls > github_wayback_urls.txt

# Filter for interesting file types
cat github_wayback_urls.txt | grep -iE "\.(json|xml|yaml|yml|conf|env|bak|
    sql|log)$"

# Filter for API endpoints
cat github_wayback_urls.txt | grep -iE "/api/|/v[0-9]+/"

# Filter for parameters (potential injection points)
cat github_wayback_urls.txt | grep "?" | sort -u

# Using waymore for broader archive coverage
python3 waymore.py -i github.com -mode U -oU github_waymore_urls.txt
```

Listing 14: Wayback URL results (sample)

```
https://github.com/api/v3/users
https://github.com/api/v3/repos
https://github.com/settings/security
https://github.com/enterprise/admin
https://github.com/login/oauth/authorize?client_id=...
```

Historical URLs frequently expose endpoints that were removed from current navigation but remain accessible on the server.

2.8 GitHub Reconnaissance

GitHub repositories — both the target's own and third-party projects — can contain leaked credentials, API keys, internal URLs, and configuration details.

Listing 15: GitHub recon techniques

```
# Search GitHub for secrets related to target
# (use GitHub's code search or the API)

# TruffleHog: scan for high-entropy strings and known patterns
trufflehog github --org=github --only-verified

# GitDorker: automated GitHub dorking
python3 GitDorker.py -t <github_token> -q github.com -d dorks/alldorks.txt

# Search for .git exposure on web servers
curl -s https://target.example.com/.git/HEAD
# If returns "ref: refs/heads/main", the .git directory is exposed

# git-dumper: extract exposed .git repositories
git-dumper https://target.example.com/.git/ ./dumped_repo

# Search for environment files
# In GitHub search: "github.com" filename:.env
# In GitHub search: "github.com" filename:config.yml password
```

High-value search patterns for GitHub code search:

- `filename:.env` – Environment variable files
- `filename:config extension:yml password` – Config files with credentials
- `filename:id_rsa` – SSH private keys
- `extension:pem private` – TLS private keys
- `AKIA` – AWS access key IDs
- `sk-` – OpenAI / Stripe secret keys

3 Phase 2: Active Recon – Subdomain Discovery

Subdomain enumeration expands the attack surface by discovering hosts under the target's root domain. This phase transitions to active reconnaissance, as some tools perform DNS queries that generate logs on the target's name servers.

3.1 Subdomain Enumeration Tools

Three primary tools are used in combination for maximum coverage:

Listing 16: Subdomain enumeration with subfinder

```
# subfinder: fast, passive subdomain discovery using multiple APIs
subfinder -d github.com -all -o github_subfinder.txt

# With specific sources
subfinder -d github.com -sources crtsh,virustotal,shodan,certspotter \
    -o github_subfinder.txt

# Silent mode for piping
subfinder -d github.com -silent
```

Listing 17: Subdomain enumeration with amass

```
# amass: comprehensive enumeration with DNS brute-forcing
# Passive enumeration only
amass enum -passive -d github.com -o github_amass_passive.txt

# Active enumeration (includes DNS brute-force)
amass enum -active -d github.com -brute -w /path/to/wordlist.txt \
    -o github_amass_active.txt

# With config file for API keys
amass enum -d github.com -config ~/.config/amass/config.ini
```

Listing 18: Subdomain enumeration with assetfinder

```
# assetfinder: lightweight, fast subdomain finder
assetfinder --subs-only github.com > github_assetfinder.txt
```

Listing 19: Combining and deduplicating results

```
# Merge all results and deduplicate
cat github_subfinder.txt github_amass_passive.txt github_assetfinder.txt | \
    sort -u > github_all_subdomains.txt

# Count unique subdomains
wc -l github_all_subdomains.txt
```

Listing 20: Combined subdomain results (sample)

```
$ wc -l github_all_subdomains.txt
247 github_all_subdomains.txt

$ head -15 github_all_subdomains.txt
api.github.com
```

```
assets-cdn.github.com
avatars.githubusercontent.com
codeload.github.com
community.github.com
docs.github.com
education.github.com
enterprise.github.com
gist.github.com
github.com
lab.github.com
pages.github.com
raw.githubusercontent.com
resources.github.com
status.github.com
```

3.2 Subdomain Validation with httpx

Not all discovered subdomains resolve or run web services. Validation identifies live hosts.

Listing 21: Validating subdomains with httpx

```
# Probe for live HTTP/HTTPS services
cat github_all_subdomains.txt | httpx -silent -status-code -title \
    -tech-detect -o github_live_hosts.txt

# With additional output fields
cat github_all_subdomains.txt | httpx -silent \
    -status-code \
    -content-length \
    -title \
    -tech-detect \
    -web-server \
    -follow-redirects \
    -o github_httpx_full.txt

# Check specific ports
cat github_all_subdomains.txt | httpx -ports 80,443,8080,8443,3000,8000 \
    -silent -o github_all_ports.txt
```

Listing 22: httpx validation output (sample)

```
https://github.com [200] [GitHub] [Ruby,Nginx]
https://api.github.com [200] [GitHub API] [Ruby]
https://gist.github.com [200] [Gist] [Ruby,Nginx]
https://docs.github.com [200] [GitHub Docs] [Next.js,Node.js]
https://education.github.com [200] [GitHub Education] [Ruby]
https://status.github.com [200] [GitHub Status] [Atlassian Statuspage]
```

The `-tech-detect` flag uses Wappalyzer signatures to fingerprint technologies, which informs later testing phases.

3.3 Resolving and Filtering Live Hosts

Listing 23: DNS resolution and filtering

```
# Resolve subdomains to IPs with dnsx
cat github_all_subdomains.txt | dnsx -silent -a -resp -o github_resolved.txt

# Filter for specific response codes (exclude 404, 403 redirects)
cat github_httpx_full.txt | grep -E "\[200\]|\[301\]|\[302\]" \
    > github_interesting_hosts.txt

# Identify unique IP addresses
cat github_resolved.txt | awk '{print $2}' | sort -u > github_unique_ips.txt
```

3.4 Virtual Host Discovery

Virtual hosts (vhosts) are websites hosted on the same IP address but responding to different `Host` headers. Discovering vhosts can reveal hidden applications.

Listing 24: Virtual host brute-forcing with ffuf

```
# Brute-force vhosts on a known IP
ffuf -u http://140.82.121.3 \
     -H "Host: FUZZ.github.com" \
     -w /usr/share/seclists/Discovery/DNS/subdomains-top1million-5000.txt \
     -fs 0 \
     -o github_vhosts.json

# Filter by response size to eliminate default responses
ffuf -u http://140.82.121.3 \
     -H "Host: FUZZ.github.com" \
     -w /usr/share/seclists/Discovery/DNS/subdomains-top1million-5000.txt \
     -fw 10 \
     -mc all
```

4 Phase 3: Active Recon – Service & Content Discovery

With a validated list of live hosts, this phase identifies specific services, open ports, hidden directories, technology stacks, and client-side code.

4.1 Port Scanning

Port scanning identifies network services running on target hosts.

Listing 25: Nmap scanning examples

```
# Quick top-ports scan
```

```
nmap -sV -sC --top-ports 1000 -oA github_nmap 140.82.121.3

# Full TCP port scan
nmap -sV -p- --min-rate 1000 -oA github_nmap_full 140.82.121.3

# UDP scan (top 100 ports, UDP is slower)
sudo nmap -sU --top-ports 100 -oA github_nmap_udp 140.82.121.3

# Script scan for specific vulnerabilities
nmap --script=http-title,http-headers,ssl-enum-ciphers -p 80,443
    140.82.121.3

# Scan multiple hosts from file
nmap -sV -iL github_unique_ips.txt -oA github_nmap_batch --top-ports 100
```

Listing 26: Nmap output (sample)

```
PORT    STATE SERVICE  VERSION
22/tcp  open  ssh      OpenSSH 8.9 (protocol 2.0)
80/tcp  open  http     nginx
443/tcp open  https    nginx
9418/tcp open  git      GitHub Enterprise
```

Listing 27: Masscan for rapid port discovery

```
# Fast SYN scan of all ports (requires root)
sudo masscan -p1-65535 140.82.121.0/24 --rate=1000 \
    -oJ github_masscan.json

# Scan specific ports across a range
sudo masscan -p80,443,8080,8443 140.82.121.0/24 --rate=500

# Pipe masscan results into nmap for service detection
sudo masscan -p1-65535 140.82.121.3 --rate=1000 -oL - | \
    awk '/^open/ {print $3}' | sort -un | \
    xargs -I{} nmap -sV -p{} 140.82.121.3
```

Note: Always respect program scope regarding port scanning. Some programs restrict scanning to specific ports (typically 80, 443) or prohibit it entirely. Rate-limit scans to avoid triggering WAF blocks or causing service disruption.

4.2 Directory and Content Brute-Forcing

Directory brute-forcing discovers hidden paths, admin panels, backup files, and unlinked content.

Listing 28: ffuf directory brute-forcing

```
# Basic directory discovery
ffuf -u https://github.com/FUZZ \
```

```
    -w /usr/share/seclists/Discovery/Web-Content/raft-medium-directories.
   txt \
    -mc 200,301,302,403 \
    -o github_dirs.json

# With recursion
ffuf -u https://github.com/FUZZ \
    -w /usr/share/seclists/Discovery/Web-Content/common.txt \
    -recursion -recursion-depth 2 \
    -mc 200,301,302 \
    -o github_dirs_recursive.json

# File extension fuzzing
ffuf -u https://github.com/FUZZ \
    -w /usr/share/seclists/Discovery/Web-Content/raft-medium-files.txt \
    -e .php,.asp,.aspx,.jsp,.html,.js,.json,.xml,.yaml,.yml,.bak,.old,.txt
   \
    -mc 200 \
    -o github_files.json

# Filter by response size to remove false positives
ffuf -u https://github.com/FUZZ \
    -w /usr/share/seclists/Discovery/Web-Content/common.txt \
    -fs 1234 \
    -mc all
```

Listing 29: gobuster directory brute-forcing

```
# Directory mode
gobuster dir -u https://github.com \
    -w /usr/share/seclists/Discovery/Web-Content/common.txt \
    -t 50 \
    -o github_gobuster.txt

# DNS subdomain mode
gobuster dns -d github.com \
    -w /usr/share/seclists/Discovery/DNS/subdomains-top1million-5000.txt \
    -t 50

# VHOST mode
gobuster vhost -u https://github.com \
    -w /usr/share/seclists/Discovery/DNS/subdomains-top1million-5000.txt
```

Listing 30: dirsearch for common paths

```
# dirsearch with default wordlist
python3 dirsearch.py -u https://github.com -e php,html,js,json \
    -t 50 --format=json -o github_dirsearch.json
```

4.3 Technology Fingerprinting

Identifying the exact technology stack guides vulnerability research and exploit selection.

Listing 31: Technology fingerprinting tools

```
# WhatWeb: detailed fingerprinting
whatweb https://github.com -v

# Multiple targets from file
whatweb -i github_live_hosts.txt --log-json=github_whatweb.json

# Wappalyzer CLI
wappalyzer https://github.com
```

Listing 32: WhatWeb output (sample)

```
https://github.com [200 OK]
  Country[US], HTTPServer[GitHub.com],
  Content-Language[en-US],
  X-Frame-Options[deny],
  Strict-Transport-Security[max-age=31536000],
  X-Content-Type-Options[nosniff],
  Ruby, Nginx
```

4.4 JavaScript File Analysis

Client-side JavaScript files often contain API endpoints, authentication logic, internal paths, secrets, and comments revealing business logic.

Listing 33: JavaScript analysis techniques

```
# Extract JS file URLs from a page
curl -s https://github.com | grep -oP 'src="[^"]*\.js[^"]*"' | \
    sed 's/src="//;s/"//' | sort -u > github_js_files.txt

# LinkFinder: extract endpoints from JS files
python3 linkfinder.py -i https://github.com -d -o cli

# Search JS files for sensitive patterns
for js in $(cat github_js_files.txt); do
    echo "=== $js ===" >> js_secrets.txt
    curl -s "$js" | grep -oiE \
        "(api_key|apikey|secret|token|password|auth|bearer)['\"]?\s*[:=]\s*[
    '\"][^'\"]{8,}" \
        >> js_secrets.txt
done

# JSParser: extract relative URLs from JS
python3 JSParser.py -u https://github.com/assets/main.js
```

```
# Custom grep for endpoints
curl -s https://github.com/assets/main.js | \
    grep -oE '"\/[a-zA-Z0-9_/.-]+"|api\.[a-zA-Z0-9_.]+' | sort -u
```

4.5 Parameter Discovery

Discovering hidden or undocumented parameters can reveal injection points and debug functionality.

Listing 34: Parameter discovery tools

```
# ParamSpider: mine parameters from web archives
python3 paramspider.py -d github.com --output github_params.txt

# Arjun: brute-force hidden parameters
arjun -u https://github.com/search -m GET -o github_arjun.json

# With custom wordlist
arjun -u https://api.github.com/search/repositories \
    -m GET \
    -w /usr/share/seclists/Discovery/Web-Content/burp-parameter-names.txt

# x8: fast parameter brute-forcing
x8 -u "https://github.com/search" -w params.txt
```

5 Phase 4: API Reconnaissance

Modern web applications rely heavily on APIs. Discovering and mapping API endpoints is critical for comprehensive testing.

5.1 API Endpoint Discovery with Burp Suite

Burp Suite acts as an intercepting proxy that captures all API calls made by the application during normal browsing.

Walkthrough:

1. Configure browser proxy to `127.0.0.1:8080` (Burp's default).
2. Navigate the target application, exercising all functionality.
3. In Burp's **Target** tab, review the site map for API endpoints.
4. Use **Engagement tools** → **Discover content** for automated path discovery.
5. In **Proxy** → **HTTP History**, filter for API calls using display filters.
6. Export discovered endpoints via **Target** → right-click → **Copy URLs in this host**.
7. Use **Repeater** to manually test discovered endpoints with modified parameters.

Key Burp features for API recon:

- **Scanner (Pro)** – Automated vulnerability scanning of discovered endpoints
- **Intruder** – Parameter fuzzing and brute-forcing
- **Extensions** – Autorize (IDOR testing), Param Miner, JWT Editor

5.2 API Testing with Postman

Postman provides a structured environment for testing discovered API endpoints.

Workflow:

1. **Import API specification** – If a Swagger/OpenAPI spec is found, import it directly into Postman (`File` → `Import` → paste URL or file).
2. **Create environment variables** – Set `base_url`, `auth_token`, and other dynamic values.
3. **Build request collections** – Organize endpoints by resource (users, repos, orgs).
4. **Test authentication flows** – Test token generation, refresh, and expiration.
5. **Write test scripts** – Use Postman's JavaScript test framework to validate responses.
6. **Run collection** – Execute all requests sequentially to test the full API surface.

Example Postman environment configuration:

Listing 35: Postman environment variables

```
{
  "base_url": "https://api.github.com",
  "auth_token": "ghp_xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",
  "org_name": "test-org",
  "repo_name": "test-repo"
}
```

5.3 API Fuzzing with Kiterunner

Kiterunner is purpose-built for API endpoint discovery, using wordlists designed for common API patterns.

Listing 36: Kiterunner API fuzzing

```
# Scan with default API wordlists
kr scan https://api.github.com -w \
    ~/wordlists/kiterunner/routes-large.kite -x 20

# Brute-force mode with text wordlist
kr brute https://api.github.com \
    -w /usr/share/seclists/Discovery/Web-Content/api/api-endpoints.txt

# Scan multiple targets
```

```
kr scan -w ~/wordlists/kiterunner/routes-large.kite \
    -l github_live_hosts.txt -x 20 -j 50
```

5.4 Swagger and OpenAPI Specification Hunting

Many applications expose their API documentation, often unintentionally.

Listing 37: Finding API documentation endpoints

```
# Common Swagger/OpenAPI paths
for path in /swagger.json /swagger/v1/swagger.json /api-docs \
    /api/swagger.json /openapi.json /v2/api-docs /v3/api-docs \
    /swagger-ui.html /swagger-ui/ /api/docs /redoc /graphql; do
    code=$(curl -s -o /dev/null -w "%{http_code}" "https://api.github.com${
    path}")
    if [ "$code" != "404" ] && [ "$code" != "000" ]; then
        echo "[${code}] https://api.github.com${path}"
    fi
done

# Search for GraphQL endpoints
for path in /graphql /graphiql /playground /api/graphql /gql; do
    code=$(curl -s -o /dev/null -w "%{http_code}" \
        -X POST "https://api.github.com${path}" \
        -H "Content-Type: application/json" \
        -d '{"query":"{ __typename }"}')
    if [ "$code" != "404" ] && [ "$code" != "000" ]; then
        echo "[${code}] GraphQL: https://api.github.com${path}"
    fi
done
```

6 Phase 5: Vulnerability Scanning

Automated vulnerability scanners identify known vulnerability patterns across discovered assets. These tools complement manual testing but do not replace it.

6.1 Nuclei

Nuclei is a template-based vulnerability scanner with a large community-maintained template library.

Listing 38: Nuclei installation and usage

```
# Installation
go install -v github.com/projectdiscovery/nuclei/v3/cmd/nuclei@latest

# Update templates
nuclei -ut
```

```
# Basic scan against a single target
nuclei -u https://github.com -o github_nuclei.txt

# Scan with specific severity
nuclei -u https://github.com -severity critical,high -o github_nuclei_high.
    txt

# Scan with specific template tags
nuclei -u https://github.com -tags cve,misconfig,exposure

# Scan multiple targets from file
nuclei -l github_live_hosts.txt -severity medium,high,critical \
    -o github_nuclei_batch.txt

# Use specific template directories
nuclei -u https://github.com \
    -t ~/nuclei-templates/http/cves/ \
    -t ~/nuclei-templates/http/exposures/

# Rate limiting to avoid WAF blocks
nuclei -u https://github.com -rl 10 -c 5

# Scan with JSON output for parsing
nuclei -l github_live_hosts.txt -jsonl -o github_nuclei.jsonl
```

Listing 39: Nuclei output (sample)

```
[INF] Using Nuclei Engine 3.x.x (templates: 7500+)
[dns-waf-detect:cloudflare] [http] [info] https://github.com
[http-missing-security-headers:x-permitted-cross-domain] [http] [info] ...
[ssl-dns-names] [ssl] [info] https://github.com
[tech-detect:nginx] [http] [info] https://github.com
```

Custom Nuclei template example:

Listing 40: Custom Nuclei template

```
id: custom-api-key-exposure
info:
  name: API Key Exposure in JavaScript
  severity: high
  tags: exposure,apikey

http:
  - method: GET
    path:
      - "{{BaseURL}}/js/app.js"
      - "{{BaseURL}}/static/js/main.js"
    matchers:
      - type: regex
        regex:
```

```
          - "(?i)(api[_-]?key|apikey|secret|token)\\s*[:=]\\s*['\"][a-zA-Z0
    -9]{16,}"
```

6.2 Nikto

Nikto performs comprehensive web server scanning for known vulnerabilities, misconfigurations, and outdated software.

Listing 41: Nikto scanning

```
# Basic scan
nikto -h https://github.com -o github_nikto.txt

# With specific tuning options
nikto -h https://github.com -Tuning 123bde -Format json \
    -o github_nikto.json

# Scan with authentication
nikto -h https://github.com -id admin:password

# Scan multiple hosts
nikto -h github_live_hosts.txt -o nikto_batch.txt
```

6.3 Burp Suite Scanner

Burp Suite Professional includes an automated scanner that tests for a wide range of web vulnerabilities.

Scanner workflow:

1. After manual browsing in Phase 4, right-click target in site map.
2. Select **Scan** → configure scan type (Crawl and Audit, Audit only).
3. Configure scan settings: speed vs. thoroughness, specific check categories.
4. Review findings in the **Dashboard** and **Issue Activity** tabs.
5. Validate findings manually using **Repeater** before reporting.
6. Export scan report in HTML or XML format.

Key scan configurations:

- **Crawl settings** – Maximum depth, form submission strategy
- **Audit settings** – SQL injection, XSS, SSRF, path traversal checks
- **Resource pool** – Concurrent request limits (respect target's rate limits)

7 Phase 6: Automation and Scripting

Manual execution of each phase is time-consuming and error-prone. Automation scripts chain tools together into repeatable pipelines.

7.1 Bash Automation: Full Recon Pipeline

Listing 42: recon.sh – Automated recon pipeline

```
#!/bin/bash
# recon.sh - Automated reconnaissance pipeline
# Usage: ./recon.sh <domain>

set -euo pipefail

DOMAIN="${1:?Usage: $0 <domain>}"
OUTDIR="./recon_${DOMAIN}_$(date +%Y%m%d)"
THREADS=50
WORDLIST="/usr/share/seclists/Discovery/DNS/subdomains-top1million-5000.txt"

mkdir -p "${OUTDIR}"/{subs,live,ports,content,vulns}

echo "[*] Starting recon for: ${DOMAIN}"
echo "[*] Output directory: ${OUTDIR}"

# Phase 1: Subdomain enumeration
echo "[+] Phase 1: Subdomain enumeration"
subfinder -d "${DOMAIN}" -all -silent > "${OUTDIR}/subs/subfinder.txt" &
assetfinder --subs-only "${DOMAIN}" > "${OUTDIR}/subs/assetfinder.txt" &
amass enum -passive -d "${DOMAIN}" -o "${OUTDIR}/subs/amass.txt" 2>/dev/null
    &
wait

# Deduplicate
cat "${OUTDIR}"/subs/*.txt | sort -u > "${OUTDIR}/subs/all_subs.txt"
echo "    Found $(wc -l < "${OUTDIR}/subs/all_subs.txt") unique subdomains"

# Phase 2: Live host detection
echo "[+] Phase 2: Probing live hosts"
cat "${OUTDIR}/subs/all_subs.txt" | httpx -silent -status-code -title \
    -tech-detect -threads "${THREADS}" -o "${OUTDIR}/live/httpx.txt"

cat "${OUTDIR}/live/httpx.txt" | awk '{print $1}' > "${OUTDIR}/live/urls.txt
    "
echo "    Found $(wc -l < "${OUTDIR}/live/urls.txt") live hosts"

# Phase 3: Port scanning (top 1000 ports on unique IPs)
echo "[+] Phase 3: Port scanning"
cat "${OUTDIR}/subs/all_subs.txt" | dnsx -silent -a -resp-only | \
    sort -u > "${OUTDIR}/ports/ips.txt"
nmap -sV --top-ports 1000 --min-rate 300 \
    -iL "${OUTDIR}/ports/ips.txt" \
    -oA "${OUTDIR}/ports/nmap_scan" 2>/dev/null

# Phase 4: Content discovery on live hosts
echo "[+] Phase 4: Content discovery"
```

```
while IFS= read -r url; do
    sanitized=$(echo "${url}" | sed 's|https\?://||;s|/|_|g')
    ffuf -u "${url}/FUZZ" \
         -w "${WORDLIST}" \
         -mc 200,301,302,403 \
         -t 30 -sf -s \
         -o "${OUTDIR}/content/${sanitized}.json" \
         2>/dev/null &
    # Limit concurrent ffuf instances
    [ "$(jobs -r | wc -l)" -ge 5 ] && wait -n
done < "${OUTDIR}/live/urls.txt"
wait

# Phase 5: Vulnerability scanning
echo "[+] Phase 5: Nuclei scanning"
nuclei -l "${OUTDIR}/live/urls.txt" \
    -severity medium,high,critical \
    -rl 20 -c 10 \
    -o "${OUTDIR}/vulns/nuclei.txt" \
    2>/dev/null

# Phase 6: Wayback URLs
echo "[+] Phase 6: Wayback URL collection"
cat "${OUTDIR}/subs/all_subs.txt" | waybackurls | sort -u \
    > "${OUTDIR}/content/wayback_urls.txt"

echo "[*] Recon complete. Results in: ${OUTDIR}"
echo "[*] Summary:"
echo "    Subdomains: $(wc -l < "${OUTDIR}/subs/all_subs.txt")"
echo "    Live hosts: $(wc -l < "${OUTDIR}/live/urls.txt")"
echo "    Nuclei findings: $(wc -l < "${OUTDIR}/vulns/nuclei.txt")"
```

7.2 Python Automation: Subdomain + httpx + Nuclei Chain

Listing 43: recon_auto.py – Python recon automation

```
#!/usr/bin/env python3
"""Automated recon pipeline: subdomain enum -> httpx -> nuclei."""

import subprocess
import sys
import os
import json
from datetime import datetime
from concurrent.futures import ThreadPoolExecutor, as_completed
from pathlib import Path


def run_cmd(cmd, description=""):
    """Run a shell command and return stdout lines."""
```

```
    print(f"  [>] {description}: {' '.join(cmd[:3])}...")
    try:
        result = subprocess.run(
            cmd, capture_output=True, text=True, timeout=600
        )
        return result.stdout.strip().splitlines()
    except subprocess.TimeoutExpired:
        print(f"  [!] Timeout: {description}")
        return []
    except FileNotFoundError:
        print(f"  [!] Tool not found: {cmd[0]}")
        return []


def subdomain_enum(domain):
    """Run multiple subdomain enumeration tools."""
    results = set()
    tools = {
        "subfinder": ["subfinder", "-d", domain, "-all", "-silent"],
        "assetfinder": ["assetfinder", "--subs-only", domain],
    }
    with ThreadPoolExecutor(max_workers=3) as executor:
        futures = {
            executor.submit(run_cmd, cmd, name): name
            for name, cmd in tools.items()
        }
        for future in as_completed(futures):
            tool_name = futures[future]
            subs = future.result()
            print(f"  [+] {tool_name}: {len(subs)} subdomains")
            results.update(subs)
    return sorted(results)


def probe_live(subdomains, outdir):
    """Probe subdomains for live HTTP services."""
    sub_file = outdir / "all_subs.txt"
    sub_file.write_text("\n".join(subdomains))

    result = run_cmd(
        ["httpx", "-l", str(sub_file), "-silent", "-status-code",
         "-title", "-tech-detect", "-json"],
        "httpx probe"
    )
    live = []
    for line in result:
        try:
            data = json.loads(line)
            live.append(data)
        except json.JSONDecodeError:
            continue
```

```
    return live


def nuclei_scan(urls_file, outdir):
    """Run nuclei on discovered live hosts."""
    return run_cmd(
        ["nuclei", "-l", str(urls_file), "-severity", "medium,high,critical"
    ,
         "-rl", "15", "-c", "5", "-jsonl",
         "-o", str(outdir / "nuclei_results.jsonl")],
        "nuclei scan"
    )


def main():
    if len(sys.argv) != 2:
        print(f"Usage: {sys.argv[0]} <domain>")
        sys.exit(1)

    domain = sys.argv[1]
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    outdir = Path(f"recon_{domain}_{timestamp}")
    outdir.mkdir(parents=True, exist_ok=True)

    print(f"[*] Recon target: {domain}")
    print(f"[*] Output: {outdir}")

    # Step 1: Subdomain enumeration
    print("\n[Phase 1] Subdomain Enumeration")
    subdomains = subdomain_enum(domain)
    print(f"  [=] Total unique subdomains: {len(subdomains)}")
    (outdir / "subdomains.txt").write_text("\n".join(subdomains))

    # Step 2: Live host probing
    print("\n[Phase 2] Live Host Probing")
    live_hosts = probe_live(subdomains, outdir)
    urls = [h.get("url", "") for h in live_hosts if h.get("url")]
    print(f"  [=] Live hosts: {len(urls)}")

    urls_file = outdir / "live_urls.txt"
    urls_file.write_text("\n".join(urls))

    # Save detailed httpx results
    with open(outdir / "httpx_results.json", "w") as f:
        json.dump(live_hosts, f, indent=2)

    # Step 3: Nuclei scanning
    print("\n[Phase 3] Vulnerability Scanning")
    nuclei_scan(urls_file, outdir)

    print(f"\n[*] Recon complete. Results saved to: {outdir}")
```

```
if __name__ == "__main__":
    main()
```

7.3 TomNomNom Tools Integration

Tom Hudson's (TomNomNom) tool suite provides lightweight, composable utilities designed for piping together in recon pipelines.

Listing 44: TomNomNom tools usage

```
# anew: append lines to a file only if they don't already exist
subfinder -d github.com -silent | anew github_subs.txt

# gf: grep for patterns (XSS, SSRF, SQLi, LFI, etc.)
cat github_wayback_urls.txt | gf xss
cat github_wayback_urls.txt | gf ssrf
cat github_wayback_urls.txt | gf sqli
cat github_wayback_urls.txt | gf redirect
cat github_wayback_urls.txt | gf lfi

# qsreplace: replace all query string values
cat github_wayback_urls.txt | grep "=" | qsreplace "FUZZ"
# Output: https://github.com/search?q=FUZZ&type=FUZZ

# unfurl: extract components from URLs
cat github_wayback_urls.txt | unfurl domains
cat github_wayback_urls.txt | unfurl paths
cat github_wayback_urls.txt | unfurl keys  # query parameter names
cat github_wayback_urls.txt | unfurl values

# Chaining: find unique parameters across all wayback URLs
cat github_wayback_urls.txt | unfurl keys | sort | uniq -c | sort -rn

# Chaining: generate XSS test URLs from wayback data
cat github_wayback_urls.txt | grep "=" | qsreplace '"><script>alert(1)</
    script>' \
    | httpx -silent -mc 200 -mr "alert\(1\)"
```

7.4 Cristi Vlad Scripts Reference

Cristi Vlad's reconnaissance automation scripts provide additional pipeline patterns for bug bounty workflows. These scripts are available on GitHub and demonstrate patterns such as:

- Automated scope validation against program rules
- Multi-tool subdomain enumeration with deduplication

- Screenshot-based visual recon with `gowitness` or `aquatone`
- Notification pipelines using `notify` for new findings
- Continuous monitoring scripts that run periodically and diff results

Listing 45: Visual recon with gowitness

```
# Take screenshots of all live hosts
gowitness file -f github_live_hosts.txt -P screenshots/ \
    --delay 3 --timeout 30

# Generate HTML report
gowitness report generate -n screenshots/report.html
```

8 Phase 7: Reporting and Documentation

Thorough documentation throughout the recon process ensures findings are reproducible and actionable.

8.1 Note-Taking with Obsidian

Obsidian is a markdown-based knowledge management tool well-suited for security research documentation.

Recommended vault structure:

```
bug-bounty-vault/
  targets/
    github/
      scope.md
      subdomains.md
      live-hosts.md
      api-endpoints.md
      findings/
        finding-001-xss.md
        finding-002-idor.md
    atlassian/
      ...
  methodology/
    recon-checklist.md
    tool-commands.md
  templates/
    finding-template.md
    recon-template.md
```

Finding template fields:

- Title, severity, CVSS score
- Affected URL/endpoint
- Steps to reproduce (with screenshots)

- Impact description
- Remediation recommendation
- Tool/technique that discovered it
- Date found, date reported, program

8.2 Organizing Findings

Findings should be categorized by:

1. **Severity** – Critical, High, Medium, Low, Informational
2. **Type** – XSS, SQLi, IDOR, SSRF, Authentication bypass, Information disclosure, etc.
3. **Asset** – Which subdomain/endpoint is affected
4. **Status** – New, Reported, Triaged, Fixed, Duplicate

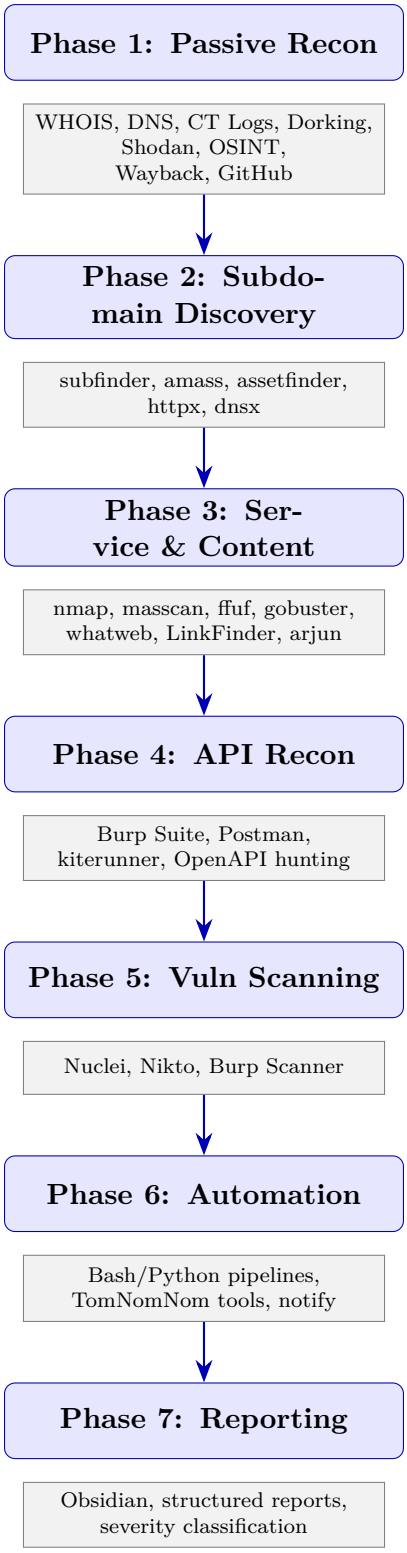
8.3 Report Structure

A professional vulnerability report includes:

1. **Title** – Clear, descriptive vulnerability title
2. **Severity** – CVSS v3.1 score with vector string
3. **Summary** – One-paragraph description of the issue
4. **Affected Asset** – Exact URL, endpoint, or component
5. **Steps to Reproduce** – Numbered steps with exact commands/requests
6. **Proof of Concept** – HTTP requests/responses, screenshots, video
7. **Impact** – What an attacker can achieve, business impact
8. **Remediation** – Specific fix recommendations
9. **References** – CVE IDs, CWE numbers, related advisories

9 Workflow Diagram

The following diagram illustrates the complete reconnaissance workflow, showing how each phase feeds into the next.



10 Tool Reference Table

Tool	Phase	Purpose	Install Command
whois	Passive	Domain registration lookup	Pre-installed on most systems

Tool	Phase	Purpose	Install Command
dig / host	Passive	DNS record queries	`apt install dnsutils`
crt.sh	Passive	Certificate transparency	Web-based / curl
Shodan CLI	Passive	Internet-wide scan database	`pip install shodan`
waybackurls	Passive	Wayback Machine URLs	`go install github.com/tomnomnom/waybackurls@late`
trufflehog	Passive	Secret scanning	`brew install trufflehog`
subfinder	Subdomain	Passive subdomain enum	`go install github.com/projectdiscovery/subfinder`
amass	Subdomain	Comprehensive enum	`go install github.com/owasp-amass/amass/v4/...@m`
assetfinder	Subdomain	Lightweight subdomain enum	`go install github.com/tomnomnom/assetfinder@late`
httpx	Validation	HTTP probe and fingerprint	`go install github.com/projectdiscovery/httpx/cmd`
dnsx	Validation	DNS resolution	`go install github.com/projectdiscovery/dnsx/cmd/`
nmap	Service	Port and service scanning	`apt install nmap`
masscan	Service	Fast port scanning	`apt install masscan`
ffuf	Content	Web fuzzer	`go install github.com/ffuf/ffuf/v2@latest`
gobuster	Content	Directory brute-forcer	`go install github.com/OJ/gobuster/v3@latest`
dirsearch	Content	Directory scanner	`pip install dirsearch`
whatweb	Fingerprint	Technology detection	`apt install whatweb`
LinkFinder	JS Analysis	Endpoint extraction	`pip install linkfinder`
arjun	Parameters	Hidden param discovery	`pip install arjun`
Burp Suite	API/Vuln	Intercepting proxy/scanner	https://portswigger.net
Postman	API	API testing platform	https://postman.com
kiterunner	API	API endpoint fuzzer	GitHub release binary
nuclei	Vuln Scan	Template-based scanner	`go install github.com/projectdiscovery/nuclei/v3`
nikto	Vuln Scan	Web server scanner	`apt install nikto`

Tool	Phase	Purpose	Install Command
anew	Utility	Deduplicated append	`go install github.com/tomnomnom/anew@latest`
gf	Utility	Pattern grep	`go install github.com/tomnomnom/gf@latest`
qsreplace	Utility	Query string replacement	`go install github.com/tomnomnom/qsreplace@latest`
unfurl	Utility	URL parsing	`go install github.com/tomnomnom/unfurl@latest`
gowitness	Utility	Screenshot tool	`go install github.com/sensepost/gowitness@latest`

Table 1: Complete tool reference with installation commands

11 Practical Demonstration: Juice Shop & Postman API Recon

To validate the methodology described in previous sections, the following walkthrough demonstrates key reconnaissance and API testing techniques against OWASP Juice Shop, a deliberately vulnerable web application designed for security training. Postman is used as the primary API recon tool, consistent with the course requirement to document API reconnaissance workflows.

All testing was performed against a local Juice Shop instance running in Docker (`docker run -p 3000:3000 bkimminich/juice-shop`), accessible at `http://localhost:3000`.

11.1 Target Discovery and Reconnaissance

11.1.1 Application Overview

The first step is to browse the target application to understand its structure, functionality, and user interface before performing any active testing.

Initial observation reveals a single-page application (SPA) built with Angular. The client-heavy architecture means the browser makes extensive API calls to a backend REST API — making browser DevTools and Postman essential for mapping the API surface.

11.1.2 API Endpoint Discovery via Browser DevTools

Opening the browser's Developer Tools (Network tab) while navigating the application reveals the underlying API calls. Every user interaction — browsing products, searching, logging in — triggers REST API requests visible in the network traffic.

By observing the network traffic, the following API patterns were identified:

- `GET /api/Products` – Product listing
- `GET /rest/products/search?q=` – Product search
- `POST /rest/user/login` – Authentication
- `POST /api/Users` – User registration
- `GET /api/Cards` – Payment card data (authenticated)

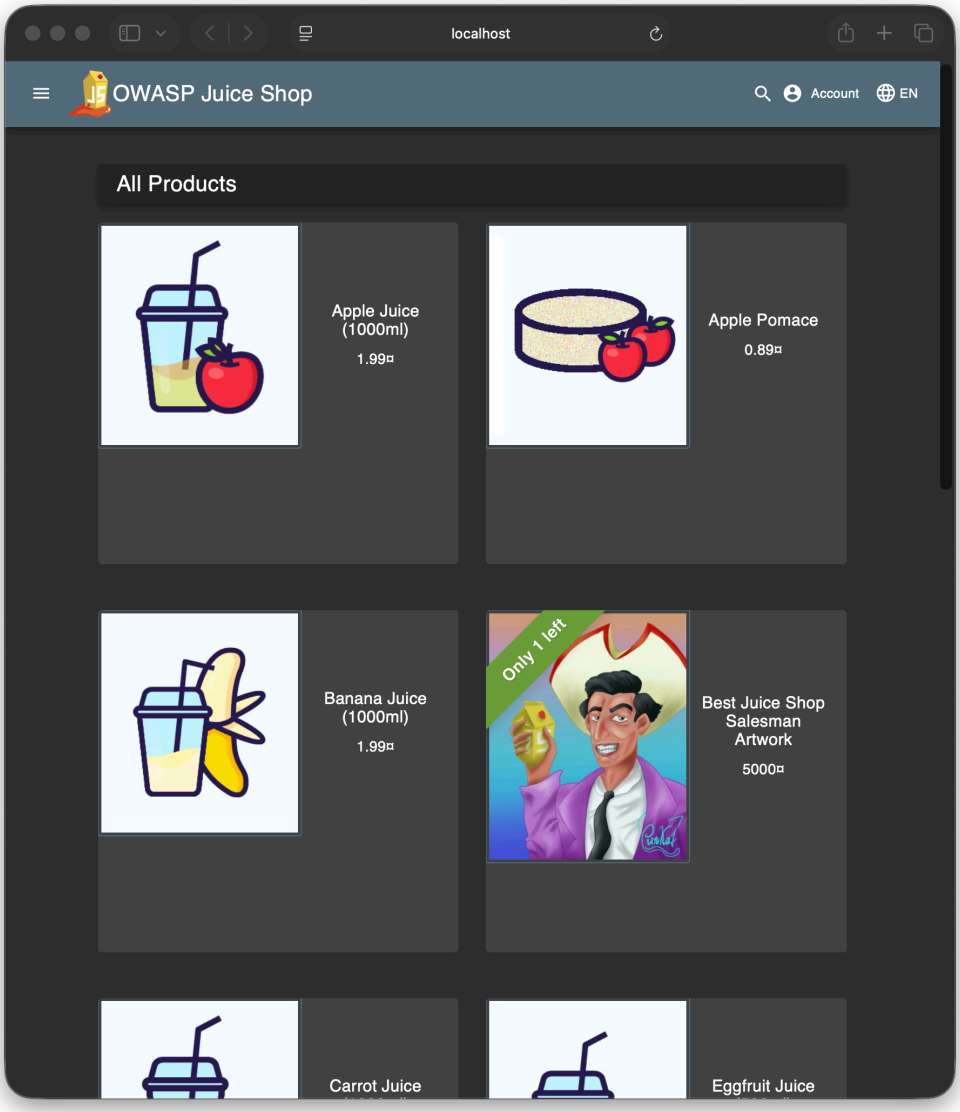


Figure 1: OWASP Juice Shop storefront running locally on port 3000

- `GET /api/Quantitys` – Inventory data

11.1.3 Examining API Responses

Inspecting individual API responses reveals the data structure, field names, and information the server returns. This informs subsequent testing by identifying potential IDOR targets (numeric IDs), sensitive data exposure, and parameter names for fuzzing.

11.2 Vulnerability Discovery

11.2.1 Hidden Content: Score Board

Juice Shop contains a hidden score board at `/#/score-board` that tracks challenge completion. Discovering this hidden route demonstrates the value of JavaScript file analysis and content discovery — the route is defined in the client-side Angular code but not linked from the UI.

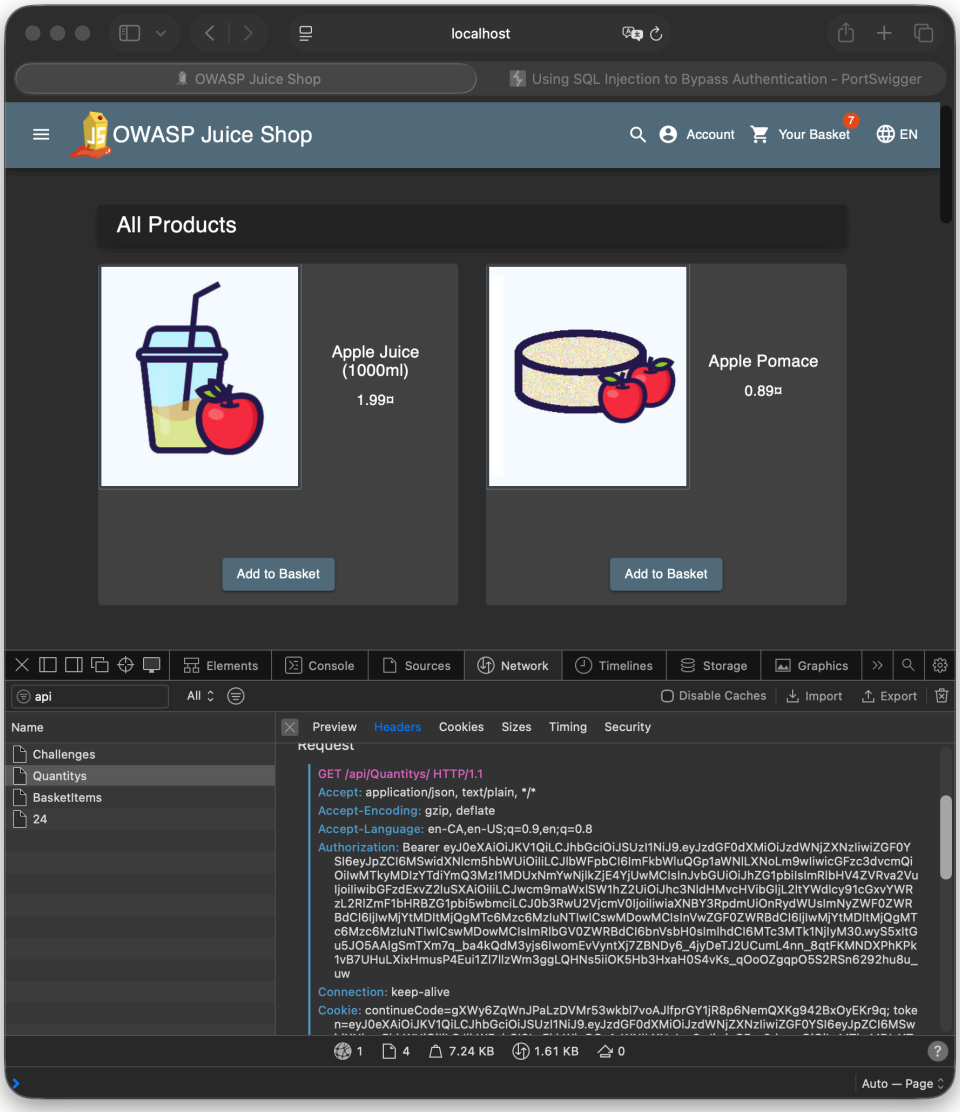


Figure 2: Browser DevTools Network tab capturing API requests during normal browsing

11.2.2 SQL Injection on Login

The login form is vulnerable to SQL injection. Entering `' OR 1=1-` as the email bypasses authentication entirely, granting access as the first user in the database (the administrator).

11.2.3 Accessing the Administration Panel

After gaining admin access via SQL injection, the administration panel at `/#/administration` becomes accessible. This panel exposes user management, feedback review, and other sensitive administrative functions.

11.3 API Reconnaissance with Postman

Postman provides a structured environment for systematically testing API endpoints discovered during the browser-based reconnaissance phase. This section demonstrates

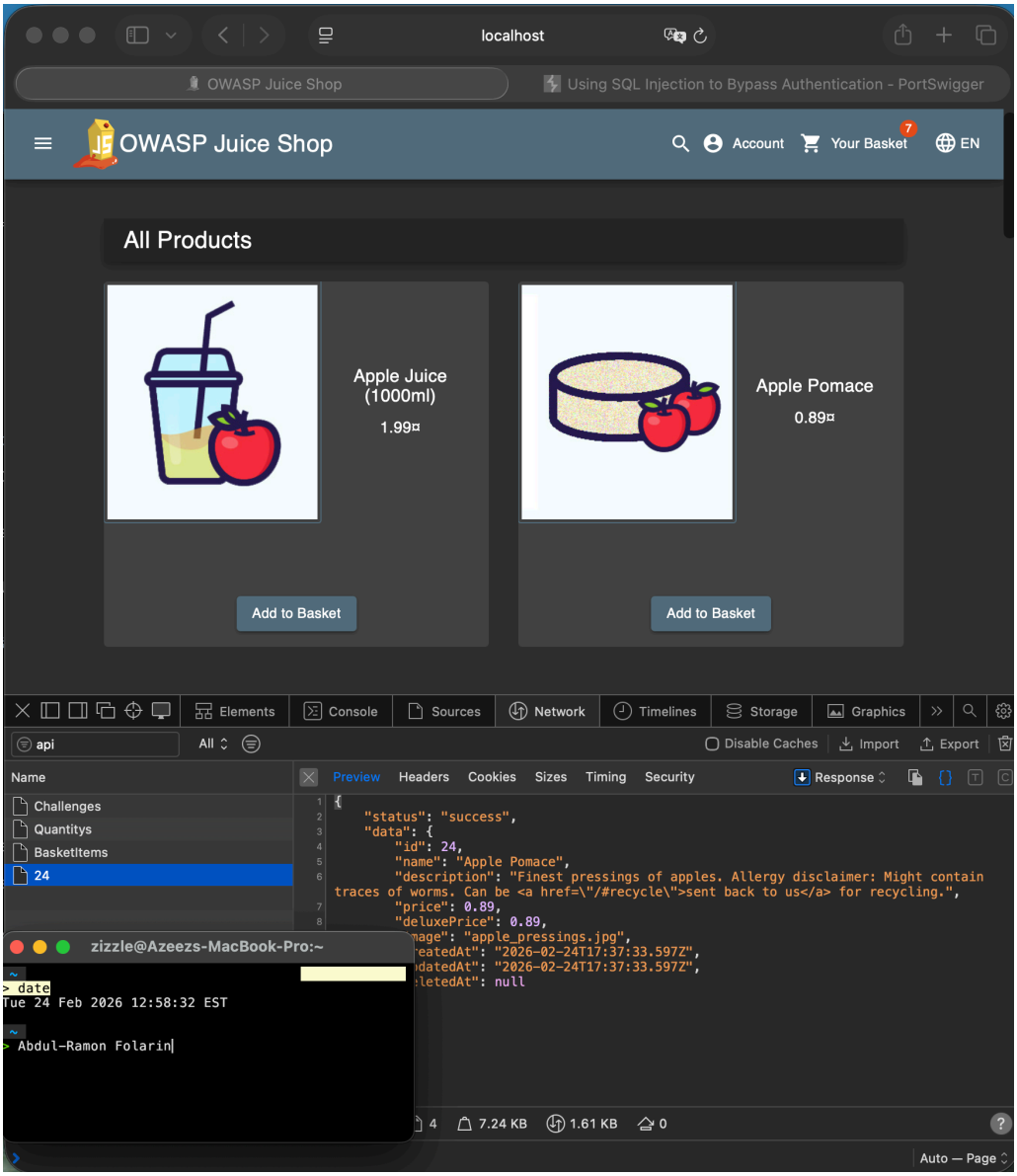


Figure 3: JSON response from `/api/Products` showing product data structure

the API recon workflow specified in the course requirements.

11.3.1 Environment Configuration

A Postman environment was created with the variable `base_url` set to `http://localhost:3000`. This allows all requests in the collection to reference `{{base_url}}` and be easily retargeted to different environments (e.g., a remote instance).

11.3.2 Collection Setup

A dedicated collection named “Juice Shop API Recon” was created to organize all discovered endpoints by category (products, authentication, users, etc.).

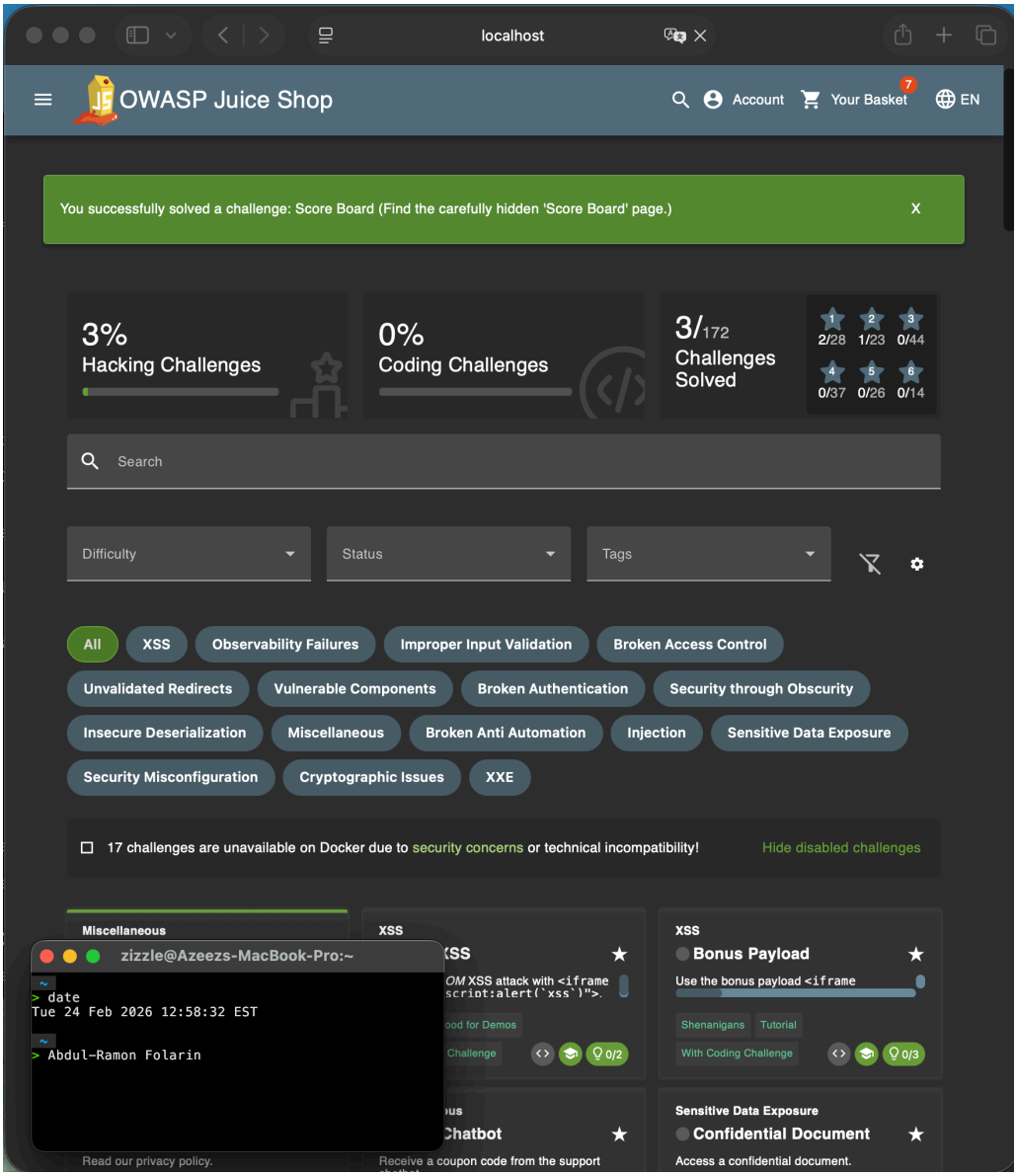


Figure 4: Hidden score board discovered at `/#/score-board`

11.3.3 Product Enumeration

The `GET /api/Products` endpoint returns the full product catalog without authentication. The response includes product IDs, names, descriptions, prices, and image paths.

11.3.4 Search Functionality Testing

The search endpoint `GET /rest/products/search?q=lemon` accepts a query parameter and returns matching products. This endpoint is a candidate for SQL injection and other injection testing, as the search term is likely interpolated into a database query.

11.3.5 Authentication: Login Flow

The `POST /rest/user/login` endpoint accepts JSON credentials and returns a JWT authentication token on success. This token is required for accessing protected endpoints.

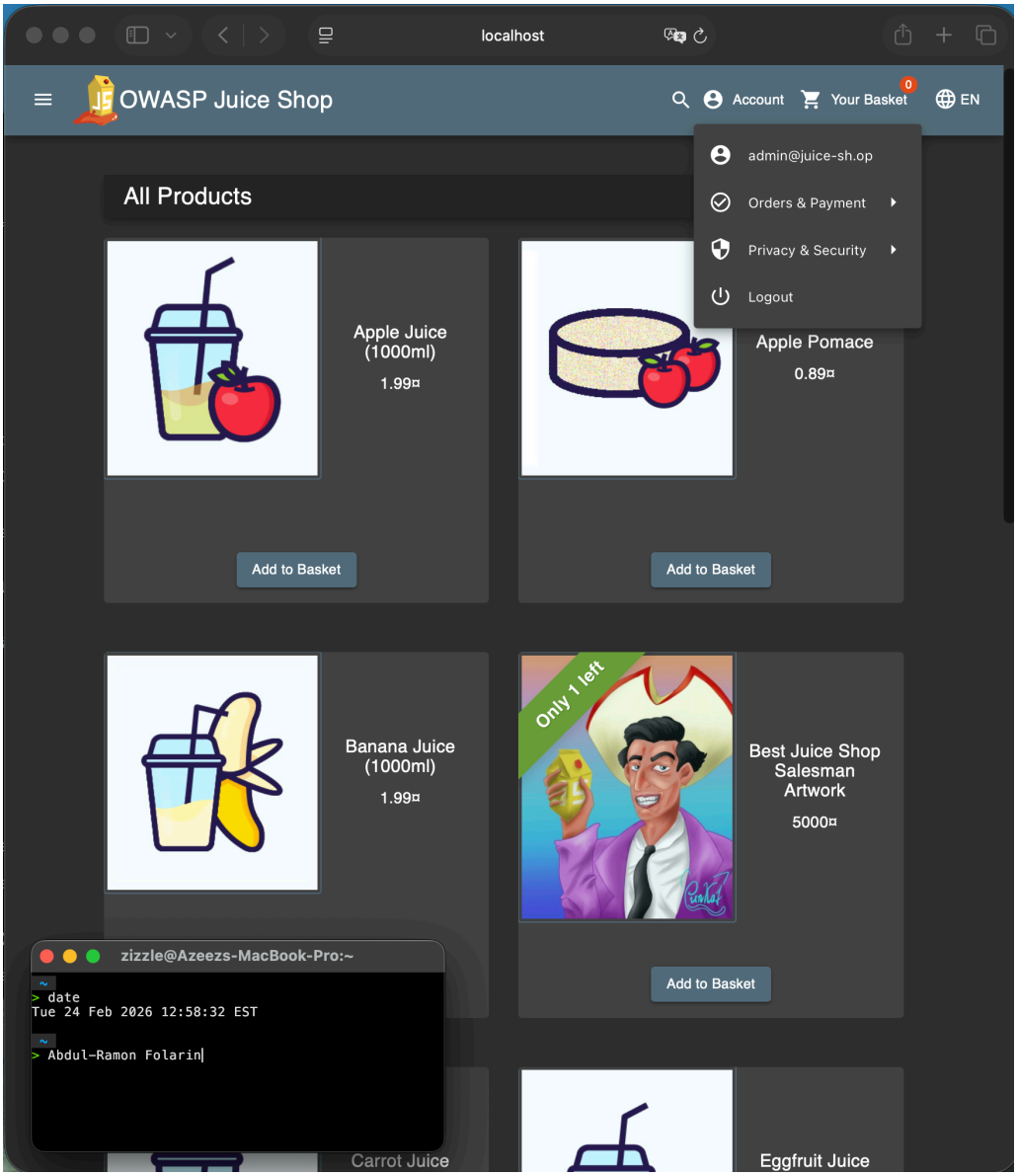


Figure 5: SQL injection payload `' OR 1=1-` used to bypass login authentication

11.3.6 Authenticated Requests

Using the JWT token obtained from the login response, authenticated API requests can be made by including an `Authorization: Bearer <token>` header. This demonstrates the standard workflow for testing protected endpoints.

11.3.7 User Registration Endpoint

The `POST /api/Users` endpoint handles user registration. Testing this endpoint reveals what data the application collects and whether input validation is properly enforced.

11.3.8 Complete API Collection

The final Postman collection contains all discovered and tested endpoints, organized for reuse. This collection serves as a reusable asset for future testing sessions and can be

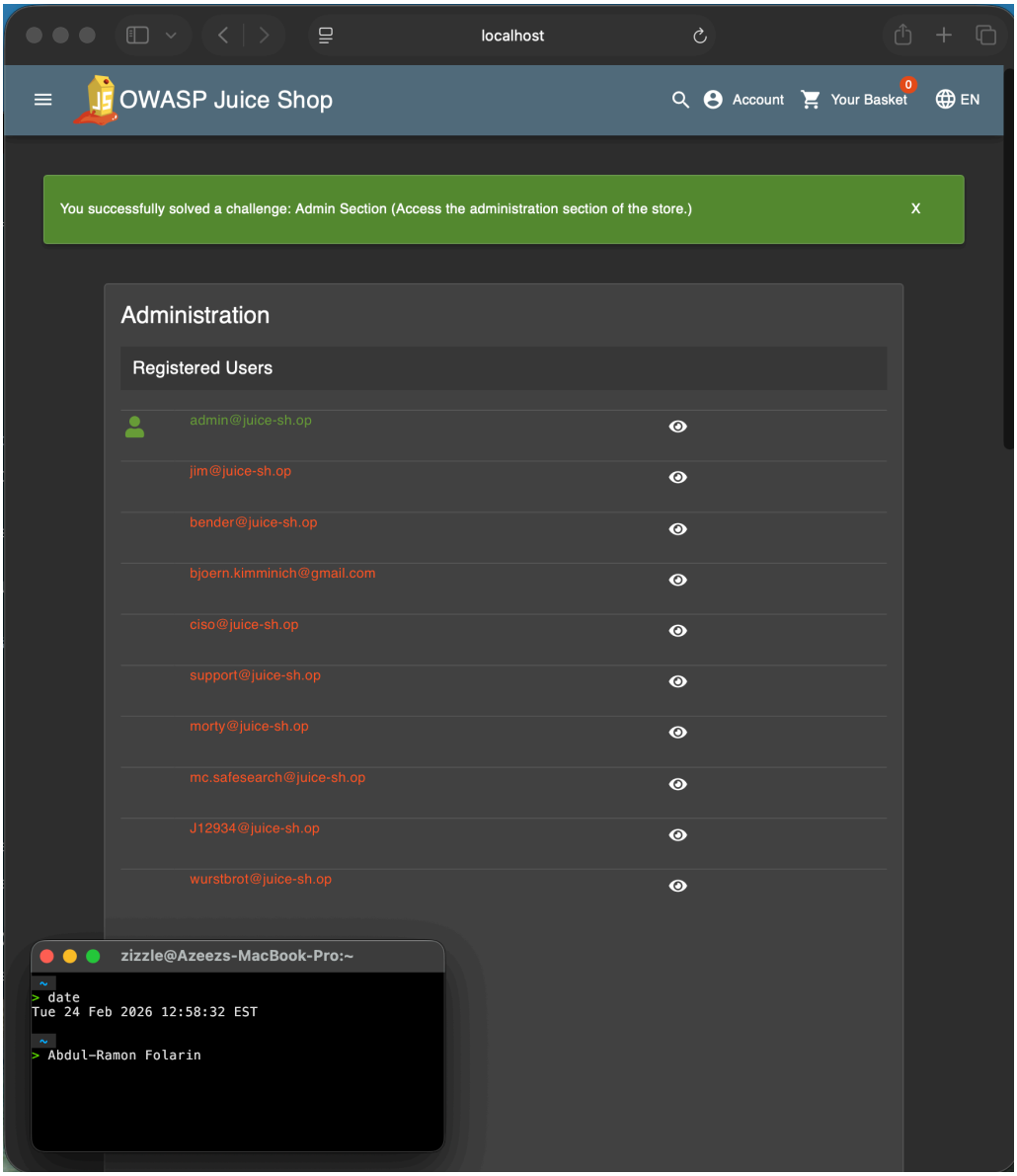


Figure 6: Administration panel accessed after SQL injection bypass

exported and shared.

11.4 Key Takeaways from Practical Exercise

The Juice Shop walkthrough demonstrates several methodology principles in practice:

1. **Passive observation first** – Browser DevTools revealed the full API surface before any active testing began.
2. **Postman as an API recon platform** – Structuring discovered endpoints into a Postman collection with environment variables creates a reusable, shareable testing asset.
3. **Authentication flow mapping** – Understanding the login → token → authenticated request chain is prerequisite to testing any protected functionality.

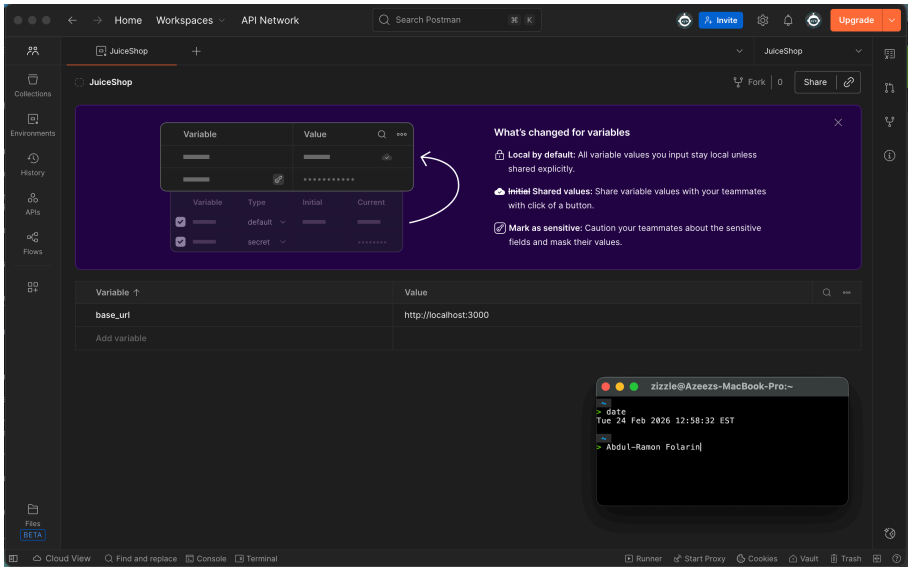


Figure 7: Postman environment configuration with Juice Shop base URL

4. **Injection testing** – The SQL injection on the login form demonstrates why input validation and parameterized queries are critical.

5. **Hidden content discovery** – The score board route, not linked from the UI, was findable through JavaScript analysis — reinforcing the value of client-side code review.

These techniques directly apply to the bug bounty reconnaissance workflow described in the preceding sections. The same Postman-based API recon approach will be used against the selected bug bounty targets (GitHub, Atlassian, Yahoo) within the scope constraints documented in Assignment 1.3.

12 Conclusion

This methodology provides a structured, repeatable approach to web application reconnaissance. By following the phased approach — from passive intelligence gathering through automated scanning — a tester can systematically discover and document the full attack surface of a target within the bounds of an authorized bug bounty program.

Key principles reinforced throughout:

- **Scope compliance** at every phase
- **Tool diversity** to maximize coverage (no single tool finds everything)
- **Automation** for repeatability and efficiency
- **Documentation** throughout the process
- **Validation** of automated results through manual review

The automation scripts in Phase 6 can be customized and extended for specific programs. As new tools and techniques emerge, they should be integrated into the appropriate phase of this methodology.

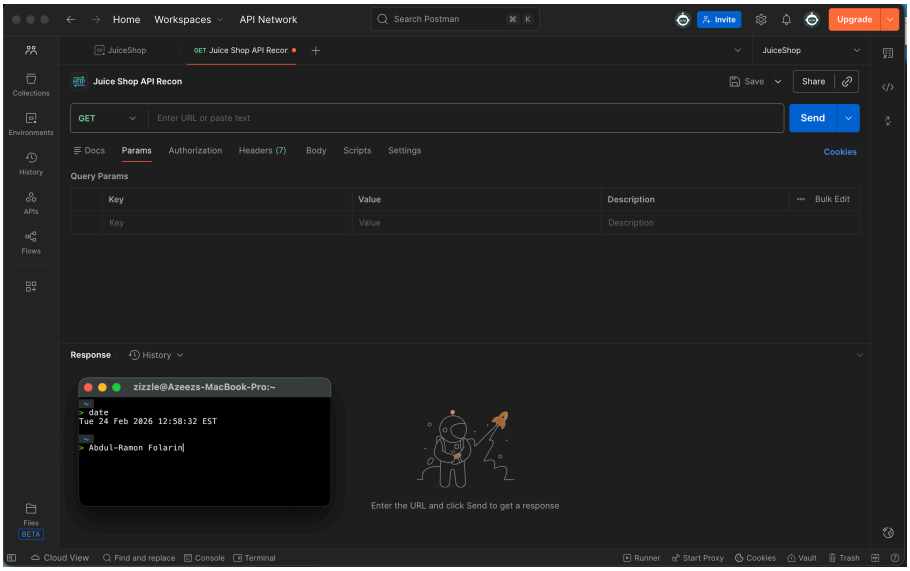


Figure 8: Postman collection created for Juice Shop API recon

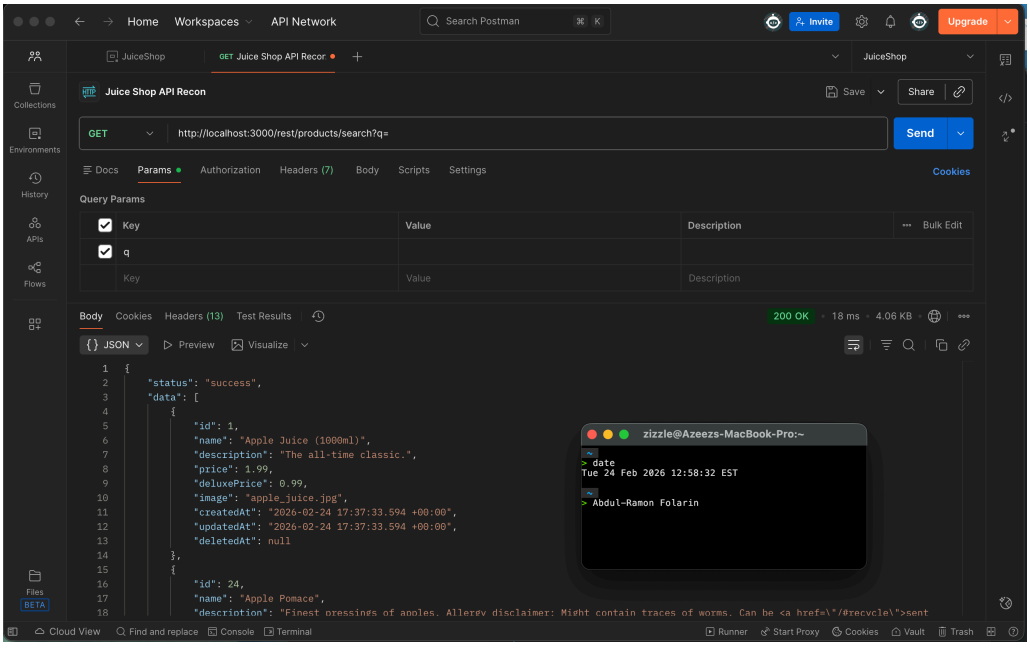


Figure 9: `GET /api/Products` – Unauthenticated product data retrieval

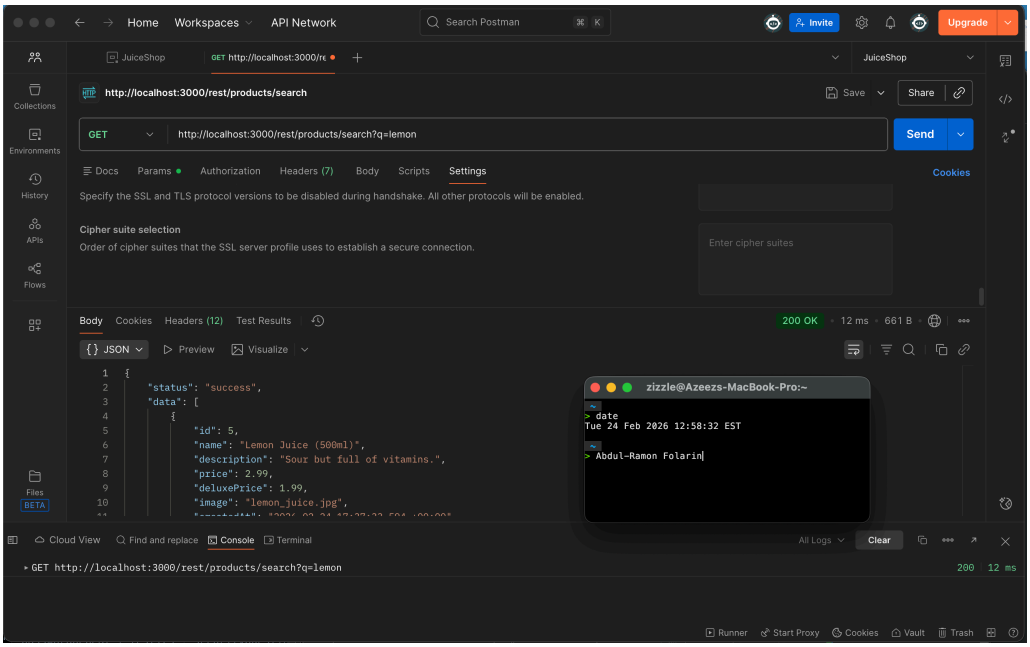


Figure 10: `GET /rest/products/search?q=lemon` – Search endpoint testing

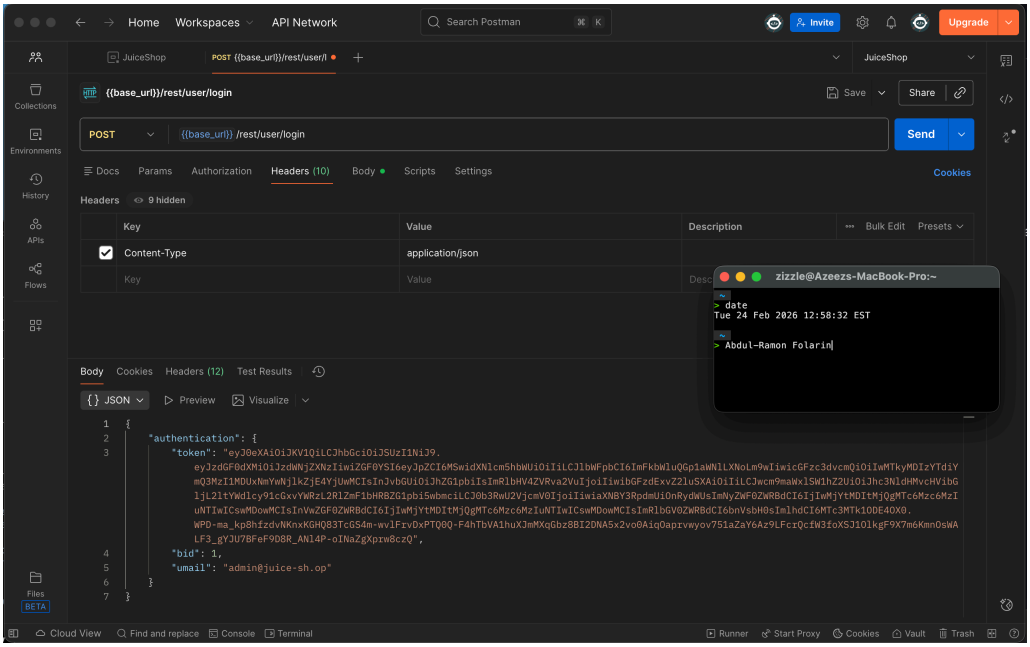


Figure 11: `POST /rest/user/login` – Authentication response with JWT token

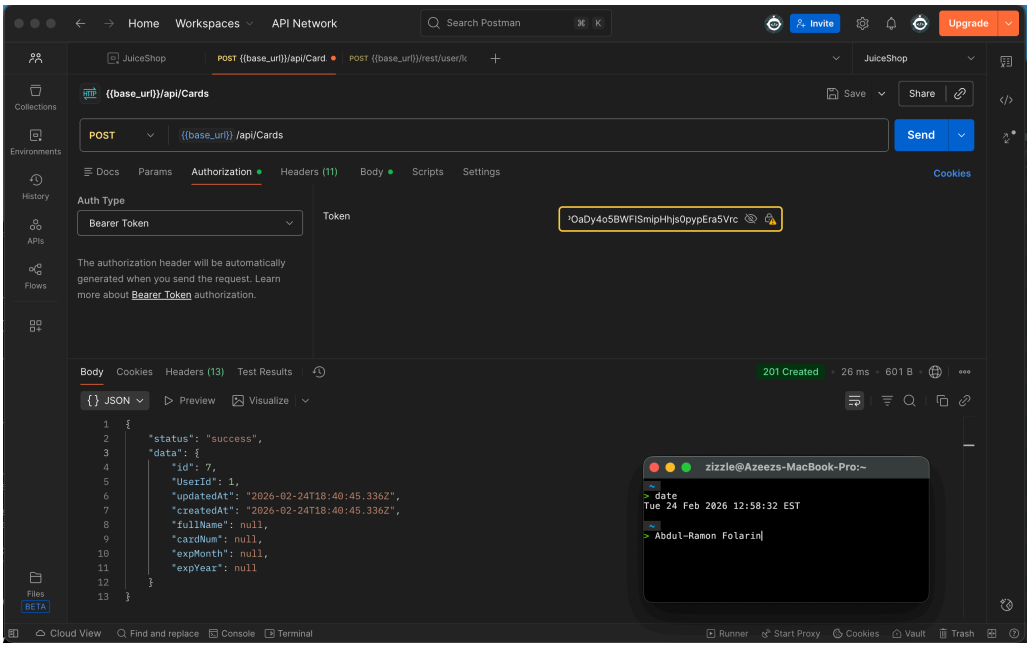


Figure 12: Authenticated request using Bearer token from login response

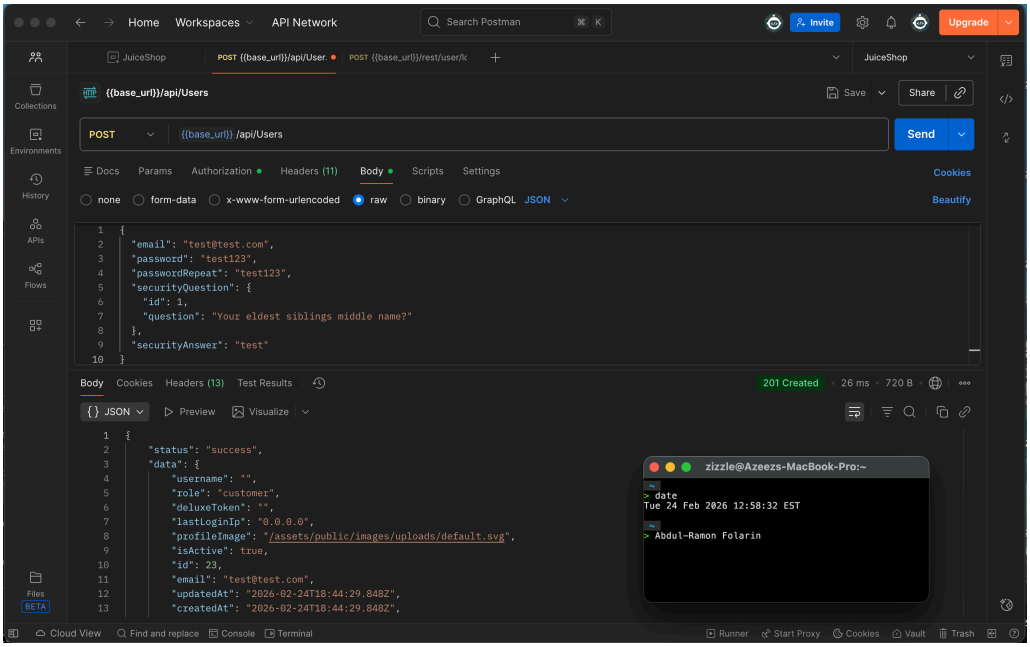


Figure 13: `POST /api/Users` – User registration with JSON payload

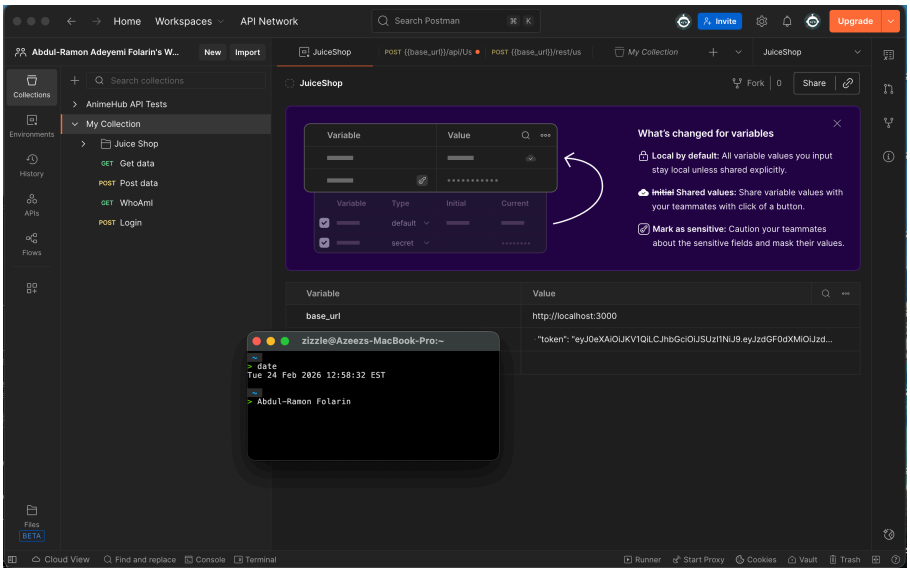


Figure 14: Complete Postman collection showing all Juice Shop API endpoints tested