Configuration Manual

MSc Research Project
CYBERSECURITY

KESAVA NAGA SIVA SAI MATLAPARTHI
Student ID: 23405210

School of Computing
National College of Ireland

Supervisor: KAMIL MAHAJAN

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Kesava Naga Siva Sai Matlaparthi
Student ID:	23405210
Programme:	CyberSecurity **Year:** 2025
Module:	MSc (Research) Practicum Part 2
Lecturer:	Kamil Mahajan
Submission Due Date:	28/01/2026
Project Title:	Context Aware and Session-Integrated Directed Fuzzing for SQL Injection Detection in Modern Web Applications
Word Count:	1470 **Page Count: 19**

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Kesava Matlaparthi

Date: 28/01/2026

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

KESAVA NAGA SIVA SAI MATLAPARTHI

Student ID:23405210

Project title: Context Aware & Session-Integrated Directed Fuzzing for SQL Injection

1. Overview

This manual explains how to set up and run the SQL injection fuzzer that was built for the MSc thesis. The tool:

- logs into a web application,
- keeps the session (cookies / tokens) alive,
- chooses up to 1–3 likely SQL injection parameters,
- sends a small staged set of payloads (boolean, error-based, time-based),
- checks the responses for signals, and
- writes all results to CSV files for later analysis.

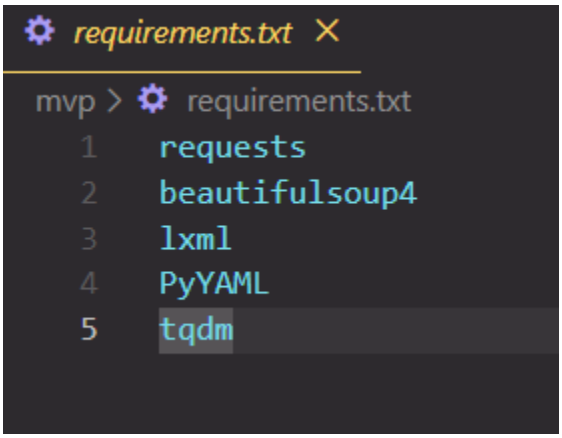
Each target (DVWA, Juice Shop) is described by a YAML file, for example:

- target_dvwa.yaml
- target_juice.yaml

2. Requirements

2.1. Software

- Python: version 3.10 or newer
- Python packages: install with pip install -r requirements.txt



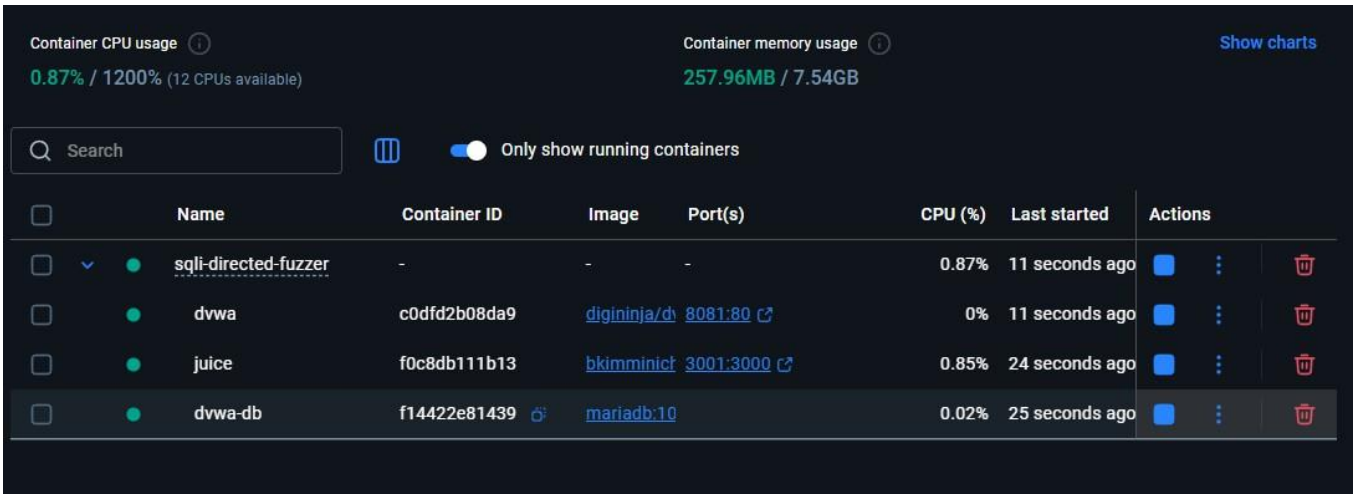
2.2. Targets

The evaluation assumes local instances of:

- Damn Vulnerable Web Application (DVWA)
- OWASP Juice Shop

Typical example URLs:

- DVWA at http://localhost:8081
- Juice Shop at http://localhost:3001



3. Main project files

Key source files in the prototype:

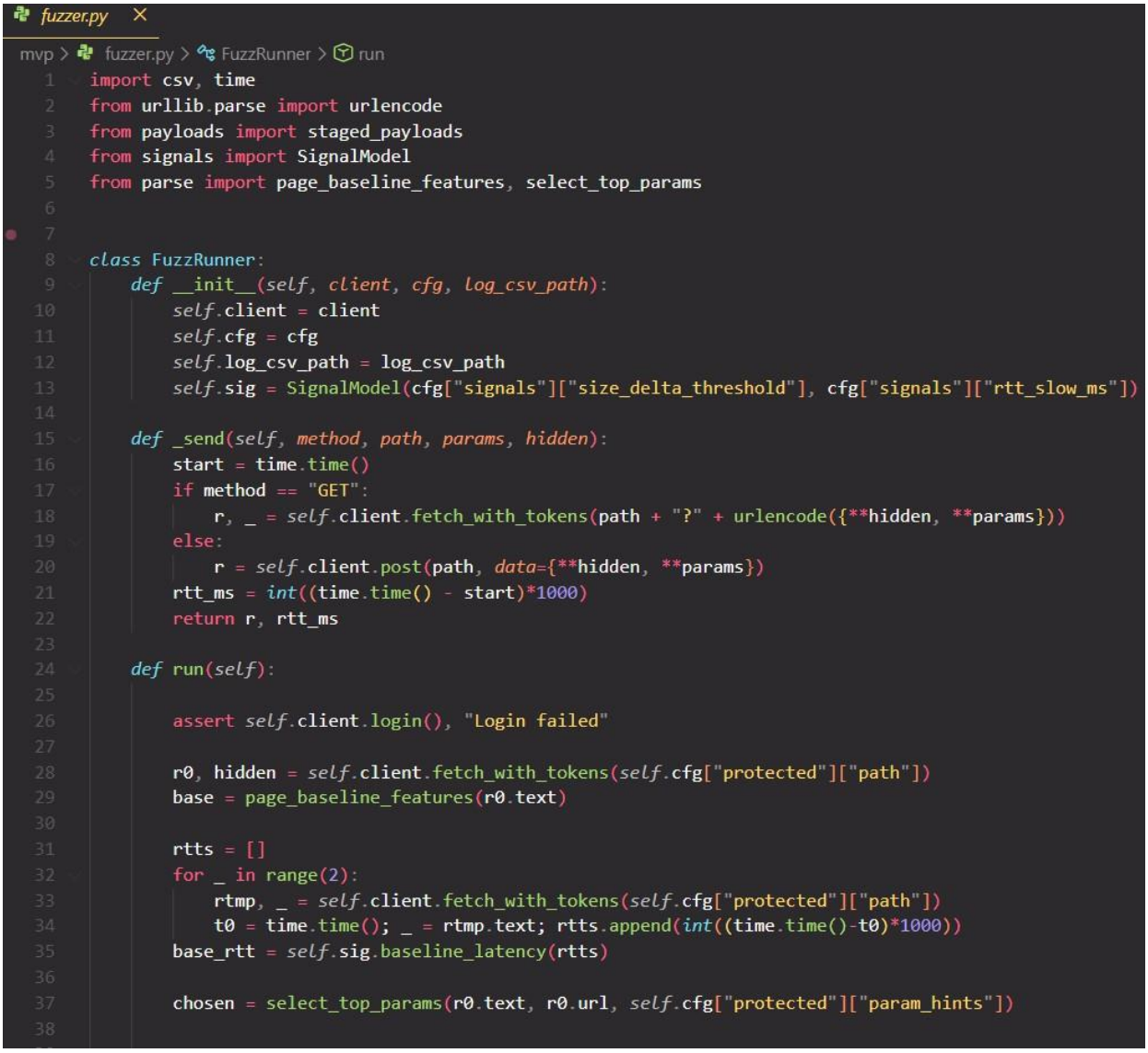
- main.py – command-line entry point. Reads a YAML configuration and starts a fuzzing run.
- session.py – defines the SessionClient class, which performs login, manages cookies, tokens and basic authentication checks.

session.py

mvp > session.py > SessionClient

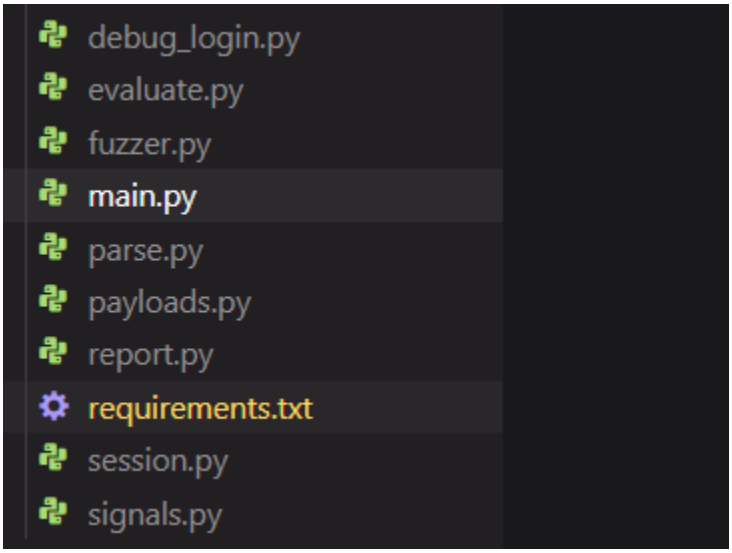
```python
import requests
from bs4 import BeautifulSoup as BS
from urllib.parse import urljoin, urlparse

class SessionClient:
    def __init__(self, base_url: str, cfg: dict):
        self.base = base_url.rstrip("/")
        self.s = requests.Session()
        self.s.headers.update({
            "User-Agent": "mvp-fuzzer/0.5 (+win)",
            "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,application/json;q=0.8,*/*;q=0.7",
        })
        self.cfg = cfg

    def _abs(self, path: str) -> str:
        return urljoin(self.base + "/", path.lstrip("/"))

    def get(self, path, **kw):
        kw.setdefault("allow_redirects", True)
        return self.s.get(self._abs(path), **kw)

    def post(self, path, data=None, **kw):
        kw.setdefault("allow_redirects", True)
        kw.setdefault("headers", {})
        kw["headers"].setdefault("Content-Type", "application/x-www-form-urlencoded")
        return self.s.post(self._abs(path), data=data, **kw)

    def post_json(self, path, json_data: dict, **kw):
        kw.setdefault("allow_redirects", True)
        kw.setdefault("headers", {})
        kw["headers"]["Content-Type"] = "application/json"
        return self.s.post(self._abs(path), json=json_data, **kw)

    def _apply_cookie_override(self):
        ov = (self.cfg.get("login") or {}).get("cookie_override")
        if not ov: return False
        host = urlparse(self.base).hostname or "localhost"
        for part in ov.split(";"):
            part = part.strip()
            if not part or "=" not in part:
                continue
            k, v = part.split("=", 1)
            self.s.cookies.set(k.strip(), v.strip(), domain=host, path="/")
        return True
```

- fuzzer.py – defines the FuzzRunner class, which implements the main fuzzing loop (parameter selection, payload iteration, logging).



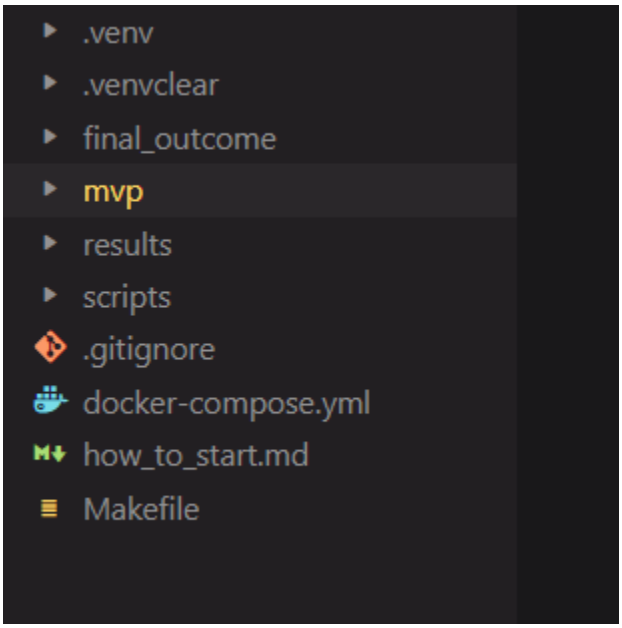
```python
import csv, time
from urllib.parse import urlencode
from payloads import staged_payloads
from signals import SignalModel
from parse import page_baseline_features, select_top_params


class FuzzRunner:
    def __init__(self, client, cfg, log_csv_path):
        self.client = client
        self.cfg = cfg
        self.log_csv_path = log_csv_path
        self.sig = SignalModel(cfg["signals"]["size_delta_threshold"], cfg["signals"]["rtt_slow_ms"])

    def _send(self, method, path, params, hidden):
        start = time.time()
        if method == "GET":
            r, _ = self.client.fetch_with_tokens(path + "?" + urlencode({**hidden, **params}))
        else:
            r = self.client.post(path, data={**hidden, **params})
        rtt_ms = int((time.time() - start)*1000)
        return r, rtt_ms

    def run(self):

        assert self.client.login(), "Login failed"

        r0, hidden = self.client.fetch_with_tokens(self.cfg["protected"]["path"])
        base = page_baseline_features(r0.text)

        rtts = []
        for _ in range(2):
            rtmp, _ = self.client.fetch_with_tokens(self.cfg["protected"]["path"])
            t0 = time.time(); _ = rtmp.text; rtts.append(int((time.time()-t0)*1000))
        base_rtt = self.sig.baseline_latency(rtts)

        chosen = select_top_params(r0.text, r0.url, self.cfg["protected"]["param_hints"])

```

- signals.py – implements the signal model used to detect content changes, SQL error patterns and unusual response times.
- payloads.py – defines the staged payload corpus (boolean, error-based and time-based payloads, plus quote-mode variants).
- parse.py – contains helpers for parsing HTML and query strings and for ranking parameters.
- evaluate.py – processes a single run log, computes metrics such as TTFF and RPF, and generates per-run plots.
- report.py – aggregates multiple runs, produces aggregate.csv and summary plots across targets.
- debug_login.py – auxiliary script for testing and debugging the DVWA login flow.
- requirements.txt – Python dependency list.



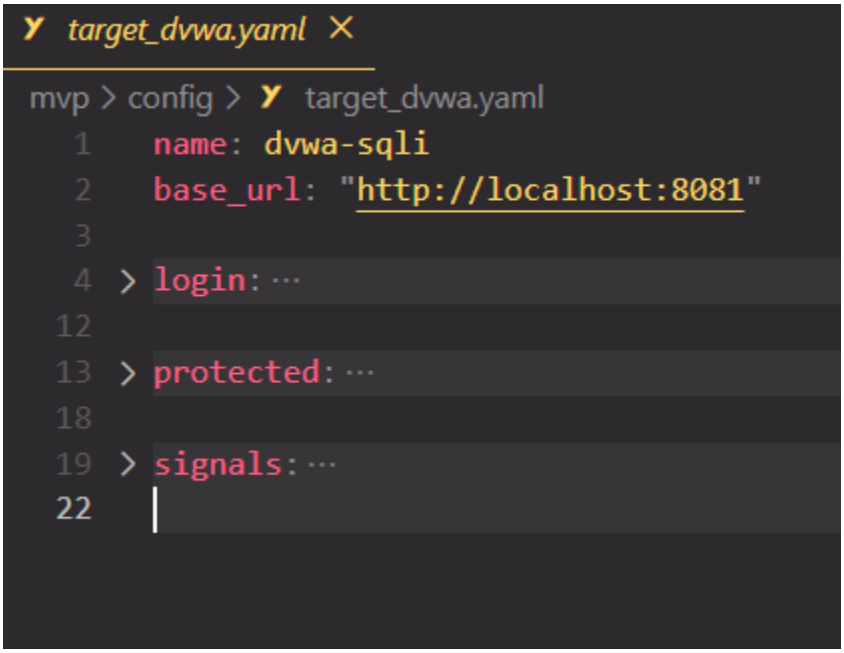
Typical output files:

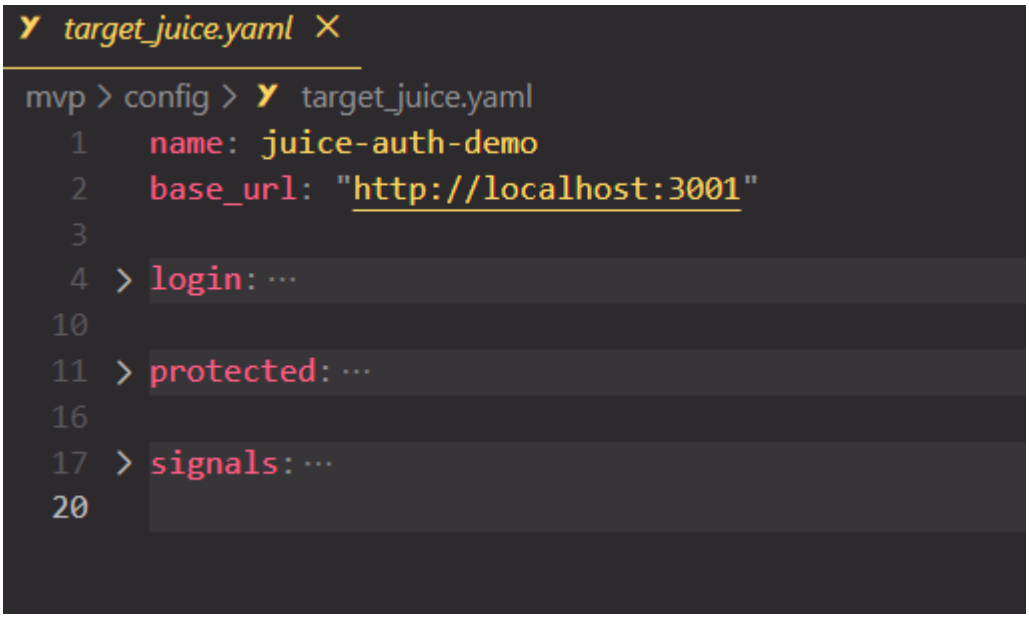
- logs/run_<name>_<timestamp>.csv – raw log for each fuzzing run.
- results/aggregate.csv – table of metrics for all runs.
- results/*.png – global figures such as TTFF boxplots, RPF histograms and attempts-vs-confirms charts.
- SUMMARY.md – convenience summary of the latest DVWA and Juice Shop runs.



4. YAML target configuration

Each target is described by a YAML file with four main parts:





The following subsections explain these parts.

4.1. Top-level fields

name

- Short identifier for the target. Used in log file names such as
- run_dvwa-sqli_1763024444.csv.

base_url

- Base URL of the application under test, without trailing slash; for example
- "http://localhost:8081".

4.2. Login block

The login block describes how to obtain an authenticated session. It is interpreted by SessionClient. Common fields:

- Path (Relative path of the login endpoint).

Examples:

- DVWA: "/login.php"
- Juice Shop: "/rest/user/login"

```
    def _juice_login_json(self) -> bool:
        login_cfg = self.cfg.get("login") or {}
        data = login_cfg.get("json") or {}
        if not data:
            return False
        resp = self.post_json(login_cfg["path"], data)
        try:
            js = resp.json()
        except Exception:
            js = {}
        token = (js.get("authentication", {}) or {}).get("token") or js.get("token")
        if token:
            self.s.headers["Authorization"] = f"Bearer {token}"
        return self._is_authed()

    def _dvwa_login_html(self) -> bool:
        login_cfg = self.cfg.get("login") or {}
        r_login = self.get(login_cfg["path"])
        soup = BS(r_login.text, "lxml")
        hidden = {i.get("name"): i.get("value","") for i in soup.select("input[type=hidden]") if i.get("name")}
        payload = {
            login_cfg.get("username_field","username"): login_cfg.get("username"),
            login_cfg.get("password_field","password"): login_cfg.get("password"),
            "Login": "Login",
            **hidden
        }
        r = self.post(login_cfg["path"], data=payload)
        if "Username and/or password" in r.text:
            return False
        return self._is_authed()
```

Form-based login (DVWA style):

- username_field – HTML form field name for the username (e.g. "username").
- password_field – HTML form field name for the password (e.g. "password").
- username – username value.
- password – password value.
- csrf_field_hint – hint for the CSRF token.

```python
def fetch_with_tokens(self, path):
    r = self.get(path)
    soup = BS(r.text, "lxml")
    hidden = {i.get("name"): i.get("value","") for i in soup.select("input[type=hidden]") if i.get("name")}
    return r, hidden
```

JSON-based login (Juice Shop style):

- json – mapping that becomes the JSON body of the login POST request. For example:

target_juice.yaml

mvp > config > target_juice.yaml

```yaml
name: juice-auth-demo
base_url: "http://localhost:3001"

login:
  path: "/rest/user/login"
  json:
    email: "admin@juice-sh.op"
    password: "admin123"
  auth_check: "/rest/user/whoami"
```

auth_check

- Relative path of a small authenticated endpoint used to confirm that login succeeded (for example /rest/user/whoami).

cookie_override

- String of cookies to inject into the session before login, such as
- "PHPSESSID=...; security=low". This is useful for setting DVWA's security level.

header_override

- Mapping of extra headers to add to every request, for example an Authorization header in setups where a static token is already known.

Login procedure (high level):

- Start a new HTTP session and apply any cookie and header overrides.
- If auth_check indicates that the session is already authenticated, skip further login steps.

- If json is present, perform a JSON POST to the login path and store the returned token or cookie.
- Otherwise, perform a form-based login: fetch the login page, collect hidden fields (including CSRF token), and send a form POST containing username, password and hidden fields.

4.3. Protected block

The protected block defines the endpoint that is fuzzed. The fields are as follows:

path

- Relative path of the protected endpoint; for example:
- DVWA SQLi module: "/vulnerabilities/sqli/"
- Juice Shop search: "/rest/products/search"

method

- Default HTTP method used when sending fuzzing requests (typically "GET" or "POST"). When a form is parsed from HTML, the form's own method may override this value for that form.

param_hints

- List of parameter names that are likely to be of interest from a SQLi perspective, such as "id", "user", "uid", "search", "q".
- These names are used in two ways:
- As hints when ranking parameters discovered in HTML forms and query strings.
- As a fallback when no parameters are visible (for example, pure JSON API endpoints).

stop_after_first_finding

- Boolean flag.
- true – stop the run as soon as a strong signal is confirmed by any oracle.
- false – continue through all staged payloads, to obtain more detailed logs and plots.

4.4. Signals block

The signals block sets thresholds for the content-delta and time-based oracles.

- size_delta_threshold
  - Minimum relative change in response length required to mark a content delta.
  - Value 0.08 corresponds to an 8% length change compared with the baseline response.
- rtt_slow_ms
  - Additional delay, in milliseconds, above the median baseline response time that counts as “slow” for the time oracle.
  - For example, 1200 corresponds to responses that are about 1.2 seconds slower than the baseline.

payloads.py

mvp > payloads.py > ...

```python
from itertools import product

BOOLEAN = [" OR 1=1 -- ", " OR 1=1#"," OR '1'='1' -- "]
ERROR   = ["'", "\"", "')" , "\" )" , "';--", "\";--"]
TIME    = ["'; WAITFOR DELAY '0:0:2'--", "'||(SELECT pg_sleep(2))--", "'; SELECT SLEEP(2); --"]

QUOTE_MODES = ["none","single","double"]

def apply_quote(s: str, mode: str):
    if mode == "single": return "'" + s
    if mode == "double": return '"' + s
    return s

def staged_payloads():
    stages = [
        ("boolean", BOOLEAN),
        ("error",   ERROR),
        ("time",    TIME),
    ]
    for family, plist in stages:
        for p, q in product(plist, QUOTE_MODES):
            yield family, q, apply_quote(p, q)
```

During initialisation, the fuzzer measures a baseline by issuing a small number of non-fuzzing requests to the protected endpoint and computing the median round-trip time.

signals.py

mvp > signals.py > ...

```python
import re, statistics, time

SQL_ERROR_RE = re.compile(
    r"(SQL syntax|SQL error|mysql|mariadb|sqlite|postgres|ORA-\d+|ODBC|unclosed quotation mark)",
    re.I
)

class SignalModel:
    def __init__(self, size_delta_threshold=0.08, rtt_slow_ms=1200):
        self.size_delta = size_delta_threshold
        self.rtt_slow   = rtt_slow_ms

    def baseline_latency(self, timings_ms):
        if not timings_ms: return 0
        return statistics.median(timings_ms)

    def content_signal(self, base_len:int, new_len:int):
        if base_len == 0: return False
        return abs(new_len - base_len)/base_len >= self.size_delta

    def error_signal(self, text:str):
        return bool(SQL_ERROR_RE.search(text or ""))

    def time_signal(self, rtt_ms:int, baseline_ms:int):
        return rtt_ms >= (baseline_ms + self.rtt_slow)
```

5. Example configs

5.1. DVWA SQLi module

target_dvwa.yaml

mvp > config > target_dvwa.yaml

```yaml
name: dvwa-sqli
base_url: "http://localhost:8081"

login:
  path: "/login.php"
  username_field: "username"
  password_field: "password"
  username: "admin"
  password: "password"
  csrf_field_hint: "user_token"
  cookie_override: "PHPSESSID=b3436d46579df9043b1b7ebb649d225b; security=low"

protected:
  path: "/vulnerabilities/sqli/"
  method: "GET"
  param_hints: ["id","user","uid","search","q"]
  stop_after_first_finding: false

signals:
  size_delta_threshold: 0.08
  rtt_slow_ms: 1200
```

This configuration targets DVWA's SQLi module at low security level, logging in via the HTML form and focusing on parameters that resemble identifiers or search fields.

5.2. Juice Shop authenticated search

target_juice.yaml

mvp > config > target_juice.yaml

```yaml
name: juice-auth-demo
base_url: "http://localhost:3001"

login:
  path: "/rest/user/login"
  json:
    email: "admin@juice-sh.op"
    password: "admin123"
  auth_check: "/rest/user/whoami"

protected:
  path: "/rest/products/search"
  method: "GET"
  param_hints: ["q","search"]
  stop_after_first_finding: false

signals:
  size_delta_threshold: 0.08
  rtt_slow_ms: 1200
```

This configuration logs in to Juice Shop using a JSON POST request, verifies authentication via /rest/user/whoami, and then fuzzes the authenticated product search endpoint.

6. Running the fuzzer

All commands in this section assume execution from the project root directory with Python 3.

Dry-run mode verifies that authentication and parameter selection behave as expected:

python main.py --target target_dvwa.yaml --dry-run

main.py

mvp > main.py > _post_run_check_for_decisions

```python
import argparse
import csv
import os
import time
import yaml
from session import SessionClient
from fuzzer import FuzzRunner


def load_cfg(path: str):
    with open(path, "r", encoding="utf-8") as f:
        return yaml.safe_load(f)


def _select_with_fallback(html: str, url: str, protected_cfg: dict):
    from parse import select_top_params
    hints = protected_cfg.get("param_hints") or []
    method = (protected_cfg.get("method") or "GET").upper()

    chosen = select_top_params(html, url, hints)
    used_fallback = False
    if not chosen:
        used_fallback = True
        chosen = [(h, method) for h in hints[:3]]

    return chosen, used_fallback


def _print_dry_run(client: SessionClient, cfg: dict):
    ok = client.login()
    assert ok, "Login failed"

    r, hidden = client.fetch_with_tokens(cfg["protected"]["path"])
    from parse import page_baseline_features
    base = page_baseline_features(r.text)

    chosen, used_fallback = _select_with_fallback(r.text, r.url, cfg["protected"])

    print("[OK] Logged in. Baseline len:", base["len"])
    print("[OK] Hidden tokens:", list(hidden.keys()))
    print("[OK] Selected params (≤3):", chosen)
    if used_fallback:
        print("[NOTE] No params discovered on page; using hints as fallback (API mode).")
```

In dry-run mode the program:

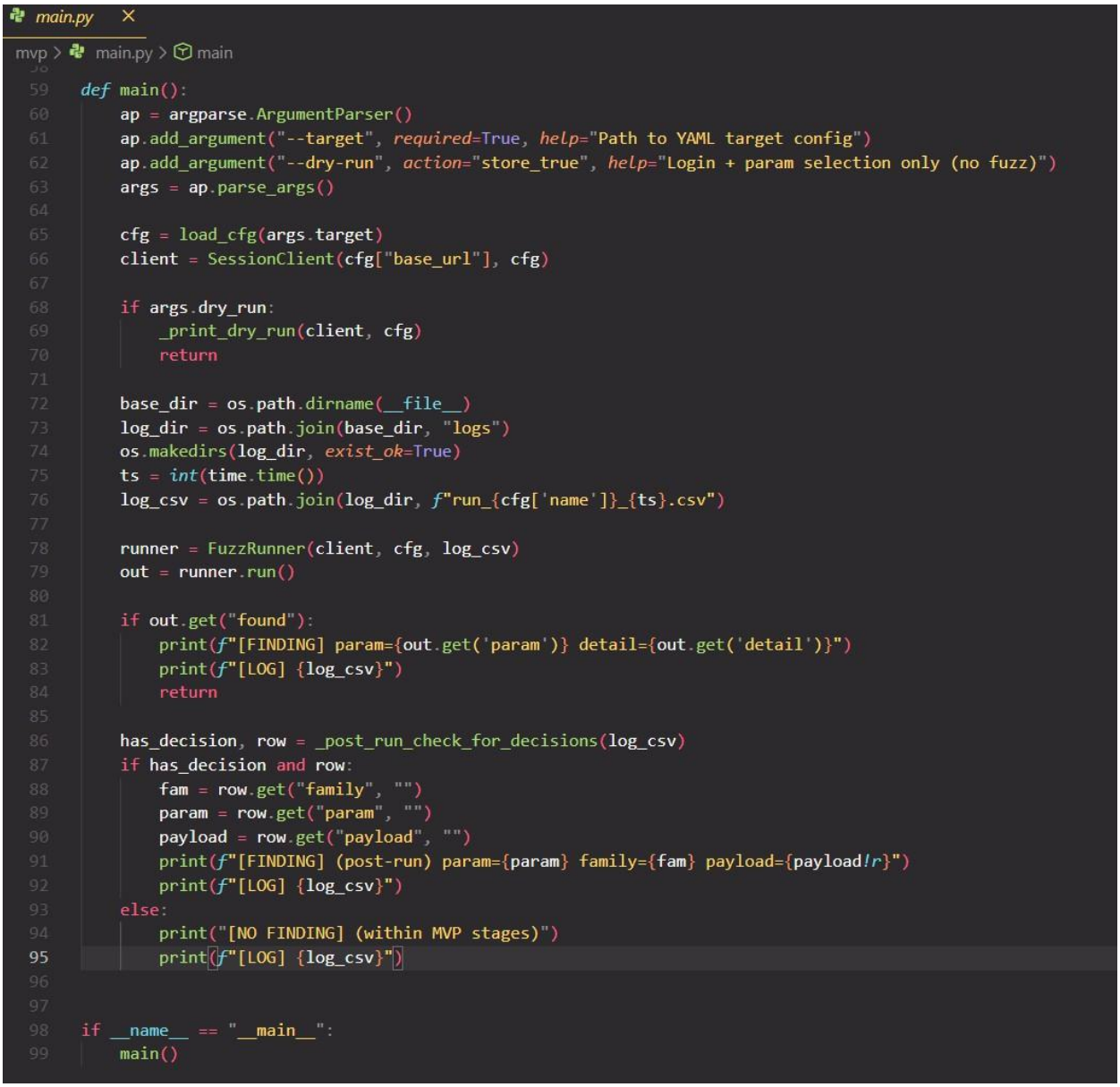
- reads the YAML configuration,
- performs the login flow,
- fetches the protected endpoint,
- measures baseline response length and time,
- selects up to three candidate parameters, and
- prints the collected information.

- No fuzzing payloads are sent in this mode.

To execute a full fuzzing run, use:

python main.py --target target_dvwa.yaml

python main.py --target target_juice.yaml



For each run:

- The YAML configuration is loaded and a SessionClient is created.
- The login flow is executed and the authenticated state is established.
- The protected endpoint is fetched and baseline metrics are gathered.
- Parameters are discovered and ranked; hints are used when necessary.

- The FuzzRunner iterates through the staged payloads (boolean, then error-based, then time-based), applying different quote modes.
- After each request, oracles compute content, error and time signals and the result is logged to the run CSV file.
- If stop_after_first_finding is true, the loop terminates after the first confirmed signal; otherwise all payloads are exercised.

The log for each run is stored as:

logs/run_<name>_<timestamp>.csv

7. Analyzing runs

A single log file can be evaluated and visualised with:

python evaluate.py --log logs/run_dvwa-sqli_1763024444.csv --plot

or, to process the latest log in a given directory:

python evaluate.py --dir logs --write-metrics --plot

evaluate.py

mvp > evaluate.py > ...

```python
import argparse, csv, glob, json, os, re, statistics
from datetime import datetime

def find_logs(logs_dir: str):
    pattern = os.path.join(logs_dir, "run_*.csv")
    return sorted(glob.glob(pattern))

def pick_latest(logs_dir: str):
    files = find_logs(logs_dir)
    if not files:
        return None
    def key(f):
        from os.path import getmtime, basename
        name, ts, _ = parse_filename_meta(f)
        return (ts or 0, getmtime(f))
    return max(files, key=key)

def parse_filename_meta(path: str):
    base = os.path.basename(path)
    m = re.match(r"run_(?P<name>.+)_(?P<ts>\d+)\.csv$", base)
    if not m:
        return (None, None, base)
    name = m.group("name")
    try:
        ts = int(m.group("ts"))
    except Exception:
        ts = None
    return (name, ts, base)

def read_rows(csv_path: str):
    with open(csv_path, "r", encoding="utf-8") as f:
        return list(csv.DictReader(f))
```

The evaluation step:

- computes total number of requests, median RTT, TTFF and RPF,
- counts attempts and confirmations per payload family, and
- generates per-run plots such as:
    - <run>_attempts.png (attempts vs confirmations by family),

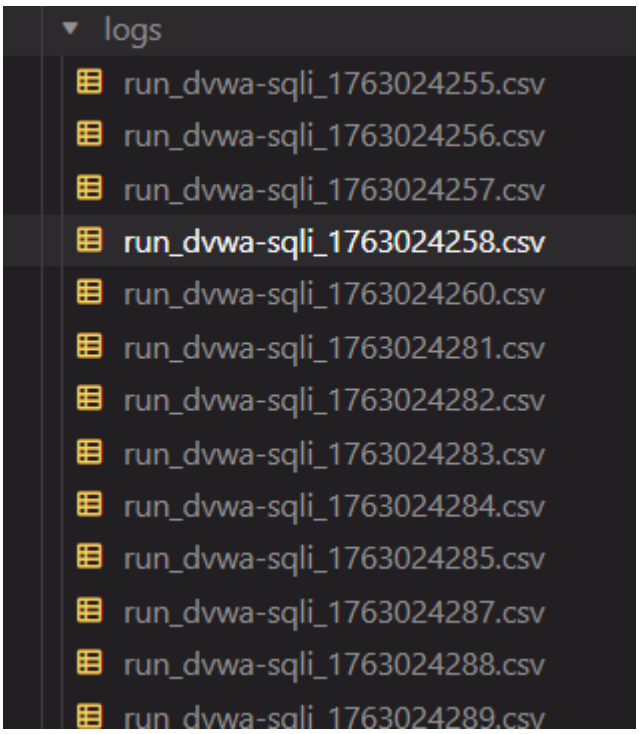
- <run>_rtt_hist.png (RTT histogram),
- <run>_cumtime.png (cumulative RTT with TTFF marker).

The flag --write-metrics causes a short summary row to be appended to a metrics CSV file. To build an aggregate report over all available logs:

python report.py

This script scans logs/run_*.csv and produces:

- results/aggregate.csv – per-run metrics for all targets,
- ttff_box_by_target.png – boxplots of TTFF grouped by target,
- rpf_hist_by_target.png – RPF histograms for each target,
- attempts_confirms_by_target.png – attempts and confirmations per payload family and target.



These artefacts are used to support the quantitative evaluation in the thesis.

8. References

Digininja. Damn Vulnerable Web Application (DVWA). Deliberately vulnerable PHP/MySQL web application for security training and tool testing.

OWASP Foundation. OWASP Juice Shop. Deliberately insecure web application for security training and capture-the-flag events.

Reitz, K., et al. Requests: HTTP for Humans. Python HTTP client library documentation and project page.

Richardson, L. Beautiful Soup. Python library for parsing and navigating HTML and XML documents.

Simonov, A., et al. PyYAML. YAML framework and parser/emitter for Python. Project and documentation.

da Costa-Luis, C., et al. tqdm. Python progress bar library for loops and iterables. Project and documentation.

Python Software Foundation. Python Language Reference, version 3.x. Official documentation for the Python programming language.