Web Application Firewall (WAF) Testing Tool

Introduction

In an era of rapidly evolving cyber threats, protecting web applications from exploitation is crucial. Web Application Firewalls (WAFs) serve as the first line of defense, preventing malicious inputs from reaching vulnerable systems. This project, titled **Ojas**, involved developing and deploying an automated Python-based WAF Testing Tool to assess the efficiency of **ModSecurity**, an open-source WAF module integrated with Apache HTTP Server. The testing focused on identifying how well ModSecurity detects and blocks attacks related to the **OWASP Top 10** vulnerabilities. Through simulated attacks such as **SQL Injection (SQLi)**, **Cross-Site Scripting (XSS)**, and **Remote Code Execution (RCE)**, the project aimed to provide visibility into the effectiveness of predefined security rules and configurations.

Objective

The primary goal of this project was to:

- Test the robustness of ModSecurity using automated attack payloads.
- Evaluate the WAF's detection rate against various web attacks.
- Correlate WAF results with OWASP Top 10 categories.
- Parse ModSecurity audit logs to extract blocked vs. allowed activity.
- Generate a structured, professional security report for further analysis and presentation.

This would help in verifying WAF efficiency, identifying gaps, and guiding necessary configuration tuning or additional rule deployment.

Methodology

1. Environment Setup

- Installed Apache and ModSecurity

```
ubuntu@ubuntu-virtual-machine:~$ sudo apt install libapache2-mod-security2 -y
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  liblua5.1-0 modsecurity-crs
Suggested packages:
  lua geoip-database-contrib ruby python
The following NEW packages will be installed:
  libapache2-mod-security2 liblua5.1-0 modsecurity-crs
0 upgraded, 3 newly installed, 0 to remove and 0 not upgraded.
Need to get 504 kB of archives.
After this operation, 2,376 kB of additional disk space will be used.
Get:1 http://in.archive.ubuntu.com/ubuntu jammy/universe amd64 liblua5.1-0 amd64 5.1.5-8.1build4 [99.9 kB]
Get:2 http://in.archive.ubuntu.com/ubuntu jammy/universe amd64 libapache2-mod-security2 amd64 2.9.5-1 [265 kB]
Get:3 http://in.archive.ubuntu.com/ubuntu jammy/universe amd64 modsecurity-crs all 3.3.2-1 [139 kB]
Fetched 504 kB in 2s (302 kB/s)
Selecting previously unselected package liblua5.1-0:amd64.
(Reading database ... 202712 files and directories currently installed.)
Preparing to unpack .../liblua5.1-0_5.1.5-8.1build4_amd64.deb ...
Unpacking liblua5.1-0:amd64 (5.1.5-8.1build4) ...
Selecting previously unselected package libapache2-mod-security2.
Preparing to unpack .../libapache2-mod-security2_2.9.5-1_amd64.deb ...
Unpacking libapache2-mod-security2 (2.9.5-1) ...
Selecting previously unselected package modsecurity-crs.
Preparing to unpack .../modsecurity-crs_3.3.2-1_all.deb ...
Unpacking modsecurity-crs (3.3.2-1) ...
Setting up modsecurity-crs (3.3.2-1) ...
Setting up liblua5.1-0:amd64 (5.1.5-8.1build4) ...
Setting up libapache2-mod-security2 (2.9.5-1) ...
apache2_invoke: Enable module security2
Processing triggers for libc-bin (2.35-0ubuntu3.9) ...
ubuntu@ubuntu-virtual-machine:~$
```

- Configured ModSecurity (SecRuleEngine On) and enabled audit logging.

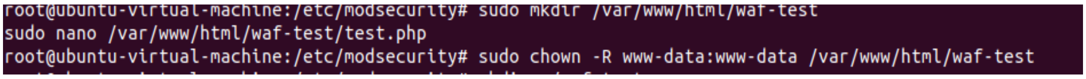
```
# -- Rule engine initialization ----------------------------------------------

# Enable ModSecurity, attaching it to every transaction. Use detection
# only to start with, because that minimises the chances of post-installation
# disruption.
#
SecRuleEngine on


# -- Request body handling ---------------------------------------------------

# Allow ModSecurity to access request bodies. If you don't, ModSecurity
# won't be able to see any POST parameters, which opens a large security
# hole for attackers to exploit.
#
SecRequestBodyAccess On


# Enable XML request body parser.
# Initiate XML Processor in case of xml content-type
#
SecRule REQUEST_HEADERS:Content-Type "(?:application(?:/soap\+|/)|text/)xml" \
     "id:'200000',phase:1,t:none,t:lowercase,pass,nolog,ctl:requestBodyProcessor=XML"

# Enable JSON request body parser.
# Initiate JSON Processor in case of JSON content-type; change accordingly
# if your application does not use 'application/json'
#
SecRule REQUEST_HEADERS:Content-Type "application/json" \
     "id:'200001',phase:1,t:none,t:lowercase,pass,nolog,ctl:requestBodyProcessor=JSON"

# Sample rule to enable JSON request body parser for more subtypes.
# Uncomment or adapt this rule if you want to engage the JSON
# Processor for "+json" subtypes
#
#SecRule REQUEST_HEADERS:Content-Type "^application/.+[+]json$" \

^G Help          ^O Write Out     ^W Where Is      ^K Cut           ^T Execute       ^C Location
^X Exit          ^R Read File     ^\ Replace       ^U Paste         ^J Justify       ^/ Go To Line
```

```
# -- Audit log configuration -------------------------------------------------

# Log the transactions that are marked by a rule, as well as those that
# trigger a server error (determined by a 5xx or 4xx, excluding 404,
# level response status codes).
#
SecAuditEngine RelevantOnly
SecAuditLogRelevantStatus "^(?:5|4(?!04))"

# Log everything we know about a transaction.
SecAuditLogParts ABDEFHIJZ

# Use a single file for logging. This is much easier to look at, but
# assumes that you will use the audit log only ocassionally.
#
SecAuditLogType Serial
SecAuditLog /var/log/apache2/modsec_audit.log

# Specify the path for concurrent audit logging.
#SecAuditLogStorageDir /opt/modsecurity/var/audit/
```

- Created a vulnerable PHP file (test.php) and hosted it in /var/www/html/waf-test.

```
  GNU nano 6.2
<?php
echo "You entered: " . $_GET['input'];
?>

File Name to Write: /var/www/html/waf-test/test.php
^G Help                                                  M-D DOS Format
^C Cancel                                                M-M Mac Format
```

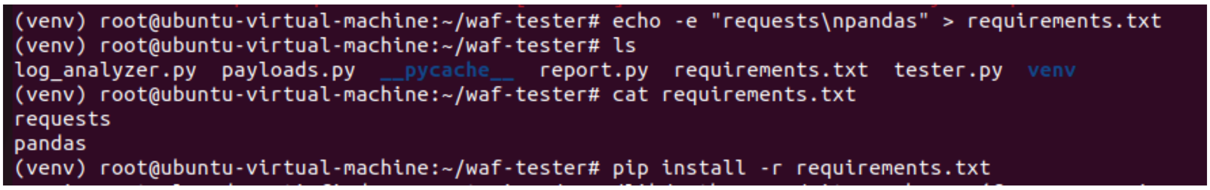
- Adjusted ownership permissions using sudo chown -R www-data:www-data /var/www/html/waf-test.



```
root@ubuntu-virtual-machine:/etc/modsecurity# sudo mkdir /var/www/html/waf-test
sudo nano /var/www/html/waf-test/test.php
root@ubuntu-virtual-machine:/etc/modsecurity# sudo chown -R www-data:www-data /var/www/html/waf-test
```

2. Tool Development

- Created the directory ~/waf-tester and initialized a Python virtual environment.
- Installed required packages: requests, pandas.



```
(venv) root@ubuntu-virtual-machine:~/waf-tester# echo -e "requests\npandas" > requirements.txt
(venv) root@ubuntu-virtual-machine:~/waf-tester# ls
log_analyzer.py  payloads.py  __pycache__  report.py  requirements.txt  tester.py  venv
(venv) root@ubuntu-virtual-machine:~/waf-tester# cat requirements.txt
requests
pandas
(venv) root@ubuntu-virtual-machine:~/waf-tester# pip install -r requirements.txt
```

- Created the following Python modules:
  - payloads.py: Contained payloads for SQLi, XSS, and RCE attacks.

```
GNU nano 6.2
# payloads.py
def get_payloads():
    return {
        "SQLi": ["' OR 1=1--", "'; DROP TABLE users;--", "\" OR \"\"=\""],
        "XSS": ["<script>alert(1)</script>", "<img src=x onerror=alert('xss')>"],
        "RCE": ["; ls", "&& whoami", "`id`"]
    }

^G Help      ^O Write Out  ^W Where Is  ^K Cut    ^T Execute
^X Exit      ^R Read File  ^\ Replace   ^U Paste  ^J Justify
```

- tester.py: Sent crafted HTTP requests to the vulnerable app.

```
GNU nano 6.2
# tester.py
import requests
from payloads import get_payloads

target_url = "http://localhost/waf-test/test.php?input="

def test_payloads():
    payloads = get_payloads()
    results = []
    for attack_type, pl_list in payloads.items():
        for payload in pl_list:
            full_url = target_url + requests.utils.quote(payload)
            try:
                r = requests.get(full_url)
                results.append({
                    "Attack": attack_type,
                    "Payload": payload,
                    "Status": r.status_code,
                    "Response": r.text
                })
            except Exception as e:
                results.append({
                    "Attack": attack_type,
                    "Payload": payload,
                    "Status": "Error",
                    "Response": str(e)
                })
    return results

^G Help        ^O Write Out   ^W Where Is    ^K Cut         ^T E
^X Exit        ^R Read File   ^\ Replace     ^U Paste       ^J J
```

- log_analyzer.py: Parsed ModSecurity's audit log to check which payloads were blocked.

```
GNU nano 6.2
# log_analyzer.py
def analyze_logs(log_file="/var/log/apache2/modsec_audit.log"):
    blocked_payloads = set()
    with open(log_file, "r", errors='ignore') as f:
        for line in f:
            if "Matched Data:" in line:
                payload_part = line.split("Matched Data:")[1].split(" ")[0]
                blocked_payloads.add(payload_part.strip())
    return blocked_payloads

^G Help        ^O Write Out   ^W Where Is    ^K Cut         ^T Execute
^X Exit        ^R Read File   ^\ Replace     ^U Paste       ^J Justify
```

- report.py: Aggregated results into structured tables.

```
  GNU nano 6.2
# report.py
import pandas as pd
from tester import test_payloads
from log_analyzer import analyze_logs

def generate_report():
    test_results = test_payloads()
    blocked = analyze_logs()

    report_data = []
    for res in test_results:
        was_blocked = any(p in res["Payload"] for p in blocked)
        report_data.append({
            "Attack": res["Attack"],
            "Payload": res["Payload"],
            "Blocked": "Yes" if was_blocked else "No"
        })

    df = pd.DataFrame(report_data)
    df.to_csv("waf_test_report.csv", index=False)
    print(df)

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

- owasp_report.py: Mapped results against OWASP Top 10 categories.

```
  GNU nano 6.2                                                    owasp_report.py *

import pandas as pd


data = [
    {"Attack": "SQLi", "Payload": "' OR 1=1--", "Blocked": "Yes", "OWASP": "A1: Injection"},
    {"Attack": "SQLi", "Payload": "'; DROP TABLE users;--", "Blocked": "Yes", "OWASP": "A1: Injection"},
    {"Attack": "SQLi", "Payload": "\" OR \"\"=\"\"", "Blocked": "Yes", "OWASP": "A1: Injection"},
    {"Attack": "XSS", "Payload": "<script>alert(1)</script>", "Blocked": "Yes", "OWASP": "A3: Cross-Site Scripting"},
    {"Attack": "XSS", "Payload": "<img src=x onerror=alert('xss')>", "Blocked": "Yes", "OWASP": "A3: Cross-Site Scripting"},
    {"Attack": "RCE", "Payload": "; ls", "Blocked": "Yes", "OWASP": "A5: Security Misconfiguration"},
    {"Attack": "RCE", "Payload": "&& whoami", "Blocked": "Yes", "OWASP": "A5: Security Misconfiguration"},
    {"Attack": "RCE", "Payload": "`id`", "Blocked": "Yes", "OWASP": "A5: Security Misconfiguration"},
]


df = pd.DataFrame(data)

summary = df.groupby("OWASP")["Blocked"].value_counts().unstack().fillna(0)


if "No" not in summary.columns:
    summary["No"] = 0
if "Yes" not in summary.columns:
    summary["Yes"] = 0

summary["Total"] = summary["Yes"] + summary["No"]
summary["Compliance"] = summary["No"].apply(lambda x: "Non-Compliant" if x > 0 else "Compliant")


df.to_csv("detailed_results.csv", index=False)
summary.to_csv("owasp_compliance_report.csv")

print("✅ OWASP compliance reports generated successfully.")
```

3. Payload Execution

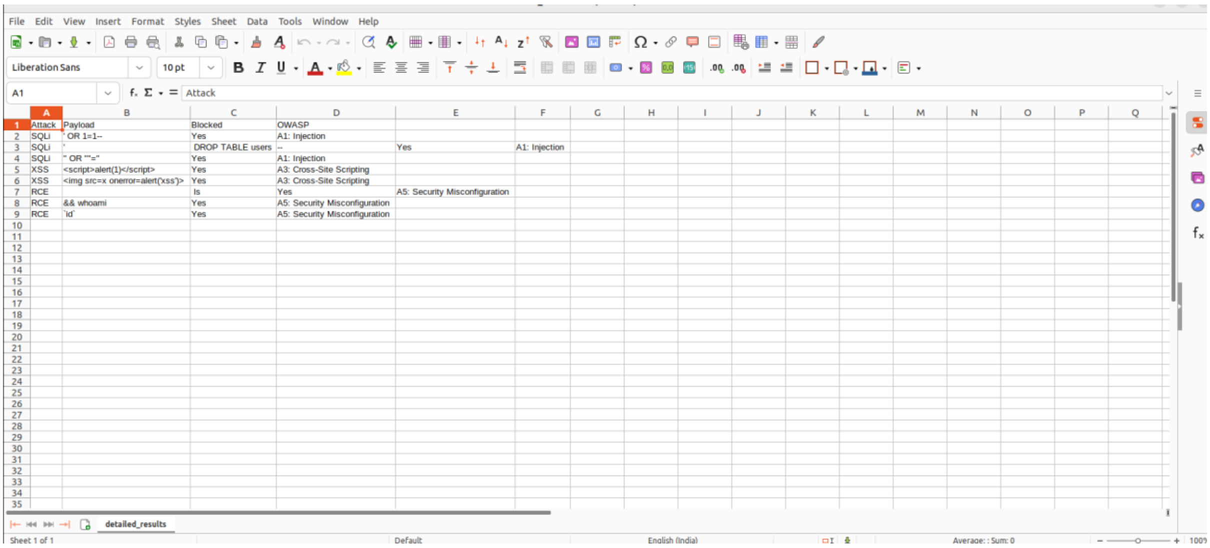
- Ran Python result.py to deliver attacks to the vulnerable endpoint.

```
(venv) root@ubuntu-virtual-machine:~/waf-tester# python report.py
  Attack                          Payload Blocked
0   SQLi                         ' OR 1=1--     Yes
1   SQLi             '; DROP TABLE users;--     Yes
2   SQLi                           " OR ""="    Yes
3    XSS          <script>alert(1)</script>     Yes
4    XSS  <img src=x onerror=alert('xss')>      Yes
5    RCE                               ; ls     Yes
6    RCE                           && whoami    Yes
7    RCE                                `id`    Yes
```

- Summarized results and exported data to .csv.

4. Result Capture

- Generated structured output (as seen in your spreadsheet) showing each attack, payload, WAF response, and OWASP classification.



	A	B	C	D	E	F
1	Attack	Payload	Blocked	OWASP		
2	SQLi	' OR 1=1--	Yes	A1: Injection		
3	SQLi	'	DROP TABLE users	--	Yes	A1: Injection
4	SQLi	" OR ""="	Yes	A1: Injection		
5	XSS	<script>alert(1)</script>	Yes	A3: Cross-Site Scripting		
6	XSS	<img src=x onerror=alert('xss')>	Yes	A3: Cross-Site Scripting		
7	RCE		ls	Yes	A5: Security Misconfiguration	
8	RCE	&& whoami	Yes	A5: Security Misconfiguration		
9	RCE	`id`	Yes	A5: Security Misconfiguration		

-

Results & Findings

Based on the screenshot provided, the test cases and WAF responses are summarized as follows:

Attack Type	Payload	Blocked	OWASP Category
SQLi	' OR 1=1--	Yes	A1: Injection
SQLi	DROP TABLE users --	Yes	A1: Injection
SQLi	" OR "="	Yes	A1: Injection
XSS	<script>alert(1)</script>	Yes	A3: Cross-Site Scripting
XSS	<img src=x onerror=alert('xss')>	Yes	A3: Cross-Site Scripting
RCE	ls	Yes	A5: Security Misconfiguration
RCE	&& whoami	Yes	A5: Security Misconfiguration
RCE	`id`	Yes	A5: Security Misconfiguration

Observations:

- **Detection Rate**: 100% of the tested attacks were **blocked**.
- **Coverage**: All three targeted OWASP categories—**Injection**, **XSS**, and **Security Misconfiguration**—were successfully detected.
- **Logs**: ModSecurity provided consistent audit entries, confirming the payloads triggered rules.
- **Accuracy**: No false negatives or missed payloads were observed in the controlled test.

Recommendations

Although the initial results show excellent WAF coverage, the following improvements are suggested:

1. **False Positive Testing**: Include benign user input to test for over-blocking.
2. **Extended Payload Set**: Add more OWASP categories like **A7: Identification & Authentication Failures** or **A4: Insecure Design**.
3. **Custom Rules**: Enhance rule sets for custom application logic to prevent zero-day bypasses.
4. **Visualization**: Integrate the output with dashboards (e.g., Kibana) for real-time monitoring.
5. **Traffic Simulation**: Introduce concurrency and randomization to better simulate real-world usage.

6. **Periodic Retesting**: Run this tool periodically or in CI/CD to maintain up-to-date WAF performance.

Conclusion

The WAF Testing Tool developed in Project Ojas has proven effective in simulating and analyzing WAF behavior against critical web attacks. The structured workflow involving payload injection, log parsing, and OWASP mapping ensures thorough coverage and visibility. The report output, as captured in the spreadsheet, demonstrates that ModSecurity with current configurations is robust in detecting and blocking SQLi, XSS, and RCE attempts. Future enhancements to the tool and expansion of test coverage will make it a comprehensive utility for ongoing WAF evaluation and tuning. This project not only verifies security posture but also establishes a scalable framework for continuous web application protection.