Business Logic Testing

Test Business Logic Data Validation (OTG-BUSLOGIC-001)

Test Objective

This test aims to verify that the application correctly validates incoming data against its expected business rules, beyond simple data type checking. The objective is to identify flaws where the application accepts logically incorrect data, such as a negative quantity for an item, a shipping date in the past, or a SQL injection payload in a user ID field. This test uses the **requests** library for HTTP communication and **BeautifulSoup** for parsing HTML forms.

Target Endpoint

The test targets various pages within the DVWA that contain forms and process user input:

[cite_start]
- `http://localhost:8080/vulnerabilities/sqli/` [cite: 2]
[cite_start]
- `http://localhost:8080/vulnerabilities/brute/` [cite: 18]
[cite_start]
- `http://localhost:8080/vulnerabilities/upload/` [cite: 32]
[cite_start]
- `http://localhost:8080/vulnerabilities/captcha/` [cite: 48]

Methodology

The testing script automates the process of submitting logically invalid data to application forms. The core logic resides in the `OTG_BUSLOGIC_001` function, which performs the following steps:

1. It iterates through a list of target URLs.
2. On each page, it uses the BeautifulSoup library to find all HTML forms.

3. For each form, it identifies all input parameters.
4. It then submits a series of predefined invalid test cases to each parameter. These test cases include negative numbers, zero values, excessively large numbers, invalid email formats, long strings, and common SQL injection payloads.
5. After each submission, the script calls the `_check_business_logic_response` helper method to analyze the server's response. This method checks if the invalid data was accepted or if an appropriate validation error was returned.

A key python snippet that checks the response is:

```
def _check_business_logic_response(self, resp, test_case):
    # ... checks for error messages ...

    # Check if the invalid value was accepted
    if test_case["value"] in resp.text:
        self.write_log(f"    [✗] Business logic validation 
failed - invalid value '{test_case['value']}' was accepted")
    else:
        self.write_log(f"    [-] No obvious business logic 
validation failure with test case '{test_case['name']}'")
```

Step to Reproduce

[cite_start]
1. The script first logs into the DVWA application and sets the security level to "low" [cite: 1].
   [cite_start]
2. It navigates to the SQL Injection test page at `http://localhost:8080/vulnerabilities/sqli/`[cite: 2].
   [cite_start]
3. The script identifies the input parameter `id` in the form[cite: 3].
4. It submits a POST request with the `id` parameter set to a logically invalid value such as `'0'` for the "zero_quantity" test case.
5. The script inspects the HTML response from the server. It finds that the invalid value '0' was reflected on the page without any error message, indicating that the input was accepted.
   [cite_start]

6. This leads to the log entry: "[✗] Business logic validation failed - invalid value '0' was accepted"[cite: 4], confirming the vulnerability. [cite_start]This process is repeated for other inputs, such as SQL injection payloads, which are also accepted[cite: 8].

Log Evidence

```
2025-08-05 02:04:57.028706 | [cite_start]----- Starting OTG-BUSLOGIC-001 (Business Logic Data Validation) ----- [cite: 2]
2025-08-05 02:04:57.028706 |
[cite_start][i] Testing URL:
http://localhost:8080/vulnerabilities/sqli/ [cite: 2]
2025-08-05 02:04:57.040322 |   - [cite_start]Testing parameter:
id [cite: 3]
2025-08-05 02:04:57.046017 |     [cite_start][-] No obvious
business logic validation failure with test case
'negative_price' [cite: 3]
2025-08-05 02:04:57.051598 |     [cite_start][✗] Business logic
validation failed - invalid value '0' was accepted [cite: 4]
2025-08-05 02:04:57.070068 |     [cite_start][✗] Business logic
validation failed - invalid value '' OR '1'='1' was accepted
[cite: 8]
...
2025-08-05 02:04:57.110423 | [cite_start][i] Testing URL:
http://localhost:8080/vulnerabilities/brute/ [cite: 18]
2025-08-05 02:04:57.110423 |   - [cite_start]Testing parameter:
username [cite: 19]
2025-08-05 02:04:57.110423 |     [cite_start][✓] Possible
validation failure with test case 'negative_price' [cite: 19]
2025-08-05 02:04:57.110423 |     [cite_start][✓] Possible
validation failure with test case 'zero_quantity' [cite: 20]
```

Test Ability to Forge Requests (OTG-BUSLOGIC-002)

Test Objective

This test is designed to determine if an attacker can manipulate or forge requests by injecting unexpected parameters. The goal is to discover hidden functionality, enable debug modes, or bypass security controls by guessing common parameter names (e.g., `debug`, `admin`, `test`) and values (e.g., `true`, `1`, `on`).

Target Endpoint

The script targets pages that are likely to have server-side logic which could be influenced by hidden parameters:

[cite_start]
- `http://localhost:8080/vulnerabilities/csrf/` [cite: 74]
[cite_start]
- `http://localhost:8080/vulnerabilities/sqli/` [cite: 204]
[cite_start]
- `http://localhost:8080/vulnerabilities/brute/` [cite: 334]
[cite_start]
- `http://localhost:8080/vulnerabilities/exec/` [cite: 464]
[cite_start]
- `http://localhost:8080/vulnerabilities/upload/` [cite: 594]

Methodology

The `OTG_BUSLOGIC_002` function systematically probes for vulnerabilities related to request forgery. The script performs the following actions:

1. It identifies all existing parameters within forms on the target pages.
2. It then iterates through a predefined list of common hidden parameter names (`debug`, `test`, `admin`, etc.).
3. For each hidden parameter name, it sends requests with various boolean-like values (`true`, `false`, `1`, `0`, `yes`, `no`, etc.).
4. The `_check_forged_response` helper method analyzes the response content for keywords like "debug", "admin", "verbose", or "internal" that would indicate the forged parameter had an effect.

The core logic for testing a forged parameter is shown below:

```
def _test_forged_requests(self, url, existing_params,
hidden_parameters, toggle_values, method="GET"):
    for param in hidden_parameters:
        if param not in existing_params:
            for value in toggle_values:
                test_params = existing_params.copy()
                test_params[param] = value

                # ... sends the request ...

                self._check_forged_response(resp, param, value)
```

Step to Reproduce

[cite_start]
1. The script logs into DVWA and begins the test[cite: 73, 74].
[cite_start]
2. It targets the CSRF page at `http://localhost:8080/vulnerabilities/csrf/`[cite: 74].
3. It crafts a new request to this page, adding a parameter that does not exist in the original form, such as `debug=true`.
4. The server's response to this forged request is analyzed. The script finds keywords indicating that a hidden feature was activated.
[cite_start]
5. This results in the log entry: "[✓] Possible admin access granted with forged param 'debug=true'"[cite: 75]. This indicates that the application insecurely processed an unexpected parameter, revealing a vulnerability. [cite_start]The test repeats this for hundreds of combinations [cite: 75-728].

Log Evidence

```
2025-08-05 02:12:23.178272 | [cite_start]----- Starting OTG-BUSLOGIC-002 (Ability to Forge Requests) ----- [cite: 74]
2025-08-05 02:12:23.178272 |
[cite_start][i] Testing URL:
http://localhost:8080/vulnerabilities/csrf/ [cite: 74]
2025-08-05 02:12:23.223890 |     [cite_start][✓] Possible admin access granted with forged param 'debug=true' [cite: 75]
```

```
2025-08-05 02:12:23.228276 |     [cite_start][✓] Possible admin
access granted with forged param 'debug=false' [cite: 76]
2025-08-05 02:12:23.232293 |     [cite_start][✓] Possible admin
access granted with forged param 'debug=1' [cite: 77]
...
2025-08-05 02:12:23.321554 |     [cite_start][✓] Possible admin
access granted with forged param 'admin=true' [cite: 105]
2025-08-05 02:12:23.326377 |     [cite_start][✓] Possible admin
access granted with forged param 'admin=false' [cite: 106]
2025-08-05 02:12:23.326892 |     [cite_start][✓] Possible admin
access granted with forged param 'admin=1' [cite: 107]
```

Test Integrity Checks (OTG-BUSLOGIC-003)

Test Objective

The objective of this test is to assess whether the application properly validates data that is supposed to be immutable, such as values in hidden form fields. An attacker could tamper with these fields to alter prices, change user privileges, or bypass business logic. This test checks for vulnerabilities by modifying existing hidden fields and injecting new, potentially impactful ones.

Target Endpoint

This test targets pages with forms that might contain hidden fields critical to the application's business logic:

[cite_start]
- `http://localhost:8080/vulnerabilities/csrf/` [cite: 732]
[cite_start]
- `http://localhost:8080/vulnerabilities/upload/` [cite: 852]
[cite_start]
- `http://localhost:8080/vulnerabilities/sqli/` [cite: 981]
[cite_start]
- `http://localhost:8080/vulnerabilities/captcha/` [cite: 1102]
[cite_start]
- `http://localhost:8080/security.php` [cite: 1231]

Methodology

The `OTG_BUSLOGIC_003` function automates the discovery of integrity check vulnerabilities through the following process:

1. It scans target pages for forms and identifies all input fields, paying special attention to those with `type="hidden"`.
2. **Test 1: Modify Existing Hidden Fields.** For each hidden field found, the script submits requests where the field's value is replaced with potentially privileged values like "admin", "root", "1", or "true".
3. **Test 2: Inject New Hidden Fields.** The script attempts to inject new, non-existent hidden fields (e.g., `user_level`, `access_level`, `privilege`) with these same privileged values.
4. The `_check_integrity_response` helper function then analyzes the server's response for keywords such as "admin", "privilege", or "access granted", which would indicate a successful bypass of integrity checks.

Step to Reproduce

[cite_start]
1. The script initiates the integrity check test after logging in[cite: 731].
[cite_start]
2. It targets a page, for example, `http://localhost:8080/vulnerabilities/upload/`[cite: 852].
[cite_start]
3. Using BeautifulSoup, it discovers a hidden field named `MAX_FILE_SIZE`[cite: 852].
4. The script then crafts a new POST request, tampering with the value of this hidden field. For instance, it sets `MAX_FILE_SIZE` to `"admin"`.
[cite_start]
5. The response is analyzed, and the script determines that the request was processed in a way that suggests the tampered value had an effect, logging: "[✓] Possible admin access granted by modifying 'MAX_FILE_SIZE' to 'admin'"[cite: 852]. This demonstrates that the server trusts and uses the client-provided hidden value without sufficient validation.

Log Evidence

```
2025-08-05 02:26:40.596614 | [cite_start]----- Starting OTG-
```

```
BUSLOGIC-003 (Integrity Checks) ----- [cite: 731]
2025-08-05 02:26:40.596614 |
[cite_start][i] Testing URL:
http://localhost:8080/vulnerabilities/csrf/ [cite: 732]
2025-08-05 02:26:40.612681 |     [cite_start][✓] Possible admin
access granted by modifying 'user_id' to 'admin' [cite: 732]
2025-08-05 02:26:40.617021 |     [cite_start][✓] Possible admin
access granted by modifying 'user_id' to 'root' [cite: 733]
...
2025-08-05 02:26:41.022519 | [cite_start][i] Testing URL:
http://localhost:8080/vulnerabilities/upload/ [cite: 852]
2025-08-05 02:26:41.028359 |   - [cite_start]Found hidden
fields: MAX_FILE_SIZE [cite: 852]
2025-08-05 02:26:41.076332 |     [cite_start][✓] Possible admin
access granted by modifying 'MAX_FILE_SIZE' to 'admin' [cite:
852]
2025-08-05 02:26:41.124644 |     [cite_start][✓] Possible admin
access granted by modifying 'MAX_FILE_SIZE' to 'root' [cite:
853]
...
2025-08-05 02:26:54.425477 | [cite_start][i] Testing URL:
http://localhost:8080/security.php [cite: 1231]
2025-08-05 02:26:54.431739 |   - [cite_start]Found hidden
fields: user_token [cite: 1231]
2025-08-05 02:26:54.481924 |     [cite_start][✓] Possible admin
access granted by modifying 'user_token' to 'admin' [cite:
1231]
```

Test for Process Timing (OTG-BUSLOGIC-004)

Test Objective

The objective of this test is to identify timing side-channel vulnerabilities. By measuring and comparing the server's response times for different inputs, an attacker may be able to infer information about the application's internal state. For example, a login attempt with a valid username might take slightly longer than one with an invalid username, allowing for user enumeration. This test focuses on identifying such discrepancies.

Target Endpoint

The test targets specific functionalities where timing differences are common:

[cite_start]
- `http://localhost:8080/vulnerabilities/brute/` (Valid vs. Invalid User) [cite: 1363, 1367]
[cite_start]
- `http://localhost:8080/vulnerabilities/sqli/` (Valid Query vs. Time-based Injection) [cite: 1371, 1375]
[cite_start]
- `http://localhost:8080/login.php` (Login Timing) [cite: 1379]

Methodology

The `OTG_BUSLOGIC_004` function implements a timing analysis attack. The process is as follows:

1. A list of test cases is defined, each containing a URL, parameters, and a description (e.g., "Valid username", "Time-based SQL injection attempt").
2. For each test case, the script sends the request 5 times to get a stable average response time, mitigating network jitter.
3. The average times for all test cases are stored.
4. Finally, the script compares the average time of each test case against every other test case. If the absolute difference in response times exceeds a predefined threshold (0.5 seconds), it is flagged as a significant timing difference.

Step to Reproduce

[cite_start]
1. The script starts the process timing test[cite: 1362].
[cite_start]
2. It first establishes a baseline by testing a valid SQL query on the SQLi page, recording an average time of approximately 0.049 seconds[cite: 1374].
[cite_start]
3. Next, it submits a time-based SQL injection payload (`1' AND sleep(2)-- `) to the same page[cite: 1375]. This query instructs the database to wait for 2 seconds before responding.
[cite_start]

4. The script measures the average response time for this payload to be approximately 2.099 seconds, which includes the 2-second sleep delay[cite: 1378].
5. The script compares the two averages and finds a difference of ~2.05 seconds. [cite_start]Since this is greater than the 0.5-second threshold, it reports a "[!] Significant timing difference"[cite: 1386, 1387], successfully identifying the time-based SQL injection vulnerability.

Log Evidence

```
2025-08-05 02:33:34.205412 | [cite_start][i] Testing URL:
http://localhost:8080/vulnerabilities/sqli/ [cite: 1371]
2025-08-05 02:33:34.205412 |   - [cite_start]Test case: Valid
SQL query [cite: 1371]
...
2025-08-05 02:33:34.457169 |   - [cite_start]Average time:
0.049 seconds [cite: 1374]
2025-08-05 02:33:34.457169 |
[cite_start][i] Testing URL:
http://localhost:8080/vulnerabilities/sqli/ [cite: 1375]
2025-08-05 02:33:34.457730 |   - [cite_start]Test case: Time-
based SQL injection attempt [cite: 1375]
2025-08-05 02:33:36.589917 |     - [cite_start]Request 1: 2.132
seconds [cite: 1375]
...
2025-08-05 02:33:44.958061 |   - [cite_start]Average time:
2.099 seconds [cite: 1378]
...
2025-08-05 02:33:45.172740 | [i] Timing comparison results:
2025-08-05 02:33:45.173326 |   [cite_start][!] Significant
timing difference (2.050s) between: [cite: 1386]
2025-08-05 02:33:45.173326 |       - [cite_start]Valid SQL
query [cite: 1386]
2025-08-05 02:33:45.173326 |       - [cite_start]Time-based SQL
injection attempt [cite: 1387]
```

Test Number of Times a Function Can be Used Limits (OTG-BUSLOGIC-005)

Test Objective

This test evaluates whether the application enforces limits on the number of times a sensitive function can be used. Functions such as login, password reset, or CAPTCHA attempts should be rate-limited to prevent abuse like brute-forcing. The objective is to repeatedly execute these functions to see if the application eventually blocks the user.

Target Endpoint

The test targets functionalities where rate-limiting is expected:

[cite_start]
- `http://localhost:8080/vulnerabilities/csrf/` (Password change) [cite: 1400]
[cite_start]
- `http://localhost:8080/security.php` (Security level change) [cite: 1404]
[cite_start]
- `http://localhost:8080/vulnerabilities/captcha/` (CAPTCHA bypass) [cite: 1409]
[cite_start]
- `http://localhost:8080/vulnerabilities/brute/` (Login attempts) [cite: 1414]

Methodology

The `OTG_BUSLOGIC_005` function and its helper, `_test_brute_force_limits`, are designed to test for the absence of rate limiting. The script executes the following logic:

1. It defines several test cases, each with a target URL, parameters, and an assumed maximum number of allowed attempts (\`max_attempts\`).
2. For each test case, it runs a loop for \`max_attempts + 2\` iterations.
3. On each iteration that exceeds \`max_attempts\`, it checks the server's response.
4. If the response indicates success (e.g., contains the word "success" or has a 200 status code without a block message), it logs a vulnerability. A proper implementation

would return a status code like 429 (Too Many Requests) or a page indicating the user is blocked.

Step to Reproduce

[cite_start]
1. The script initiates a specific test for the brute-force login page, assuming a typical limit of 5 attempts[cite: 1414].
2. It repeatedly sends POST requests with an incorrect password for the "admin" user. [cite_start]
3. After the 5th attempt, the script continues with a 6th attempt[cite: 1417].
4. It inspects the response from the 6th attempt and finds no indication of a block or lockout. The application processes the request as it did the first five. [cite_start]
5. The script logs "[✗] Still allowed after 6 attempts!"[cite: 1417], confirming that the login functionality lacks a rate-limiting mechanism. [cite_start]This is repeated for a 7th attempt with the same result[cite: 1418].

Log Evidence

```
2025-08-05 02:41:39.274906 | [cite_start][i] Testing brute force page attempt limits [cite: 1414]
2025-08-05 02:41:39.278936 |   - [cite_start]Testing brute force attempt #1 [cite: 1414]
...
2025-08-05 02:41:43.494029 |   - [cite_start]Testing brute force attempt #5 [cite: 1416]
2025-08-05 02:41:44.546126 |   - [cite_start]Testing EXCESS brute attempt #6 [cite: 1417]
2025-08-05 02:41:44.588769 |     [cite_start][✗] Still allowed after 6 attempts! [cite: 1417]
2025-08-05 02:41:45.598924 |   - [cite_start]Testing EXCESS brute attempt #7 [cite: 1418]
2025-08-05 02:41:45.644708 |     [cite_start][✗] Still allowed after 7 attempts! [cite: 1418]
```

Testing for the Circumvention of Work Flows (OTG-BUSLOGIC-006)

Test Objective

This test aims to determine if an application's business logic workflow can be bypassed. Applications often expect users to follow a specific sequence of steps (e.g., view item -> add to cart -> checkout). This test attempts to skip steps or access pages directly to see if the application correctly enforces the intended workflow.

Target Endpoint

The test targets multi-step processes within the application:

[cite_start]
- `http://localhost:8080/vulnerabilities/csrf/` (Password change workflow) [cite: 1426]
[cite_start]
- `http://localhost:8080/security.php` (Security level change workflow) [cite: 1427]
[cite_start]
- `http://localhost:8080/vulnerabilities/upload/` (File upload workflow) [cite: 1428]

Methodology

The `OTG_BUSLOGIC_006` function uses several helper methods to test different workflows. The general approach is to simulate an attacker who knows the endpoint for a later step in a process and tries to access it directly.

[cite_start]
- **`_test_password_change_workflow`**: Attempts to POST a new password directly to the change password endpoint without a valid CSRF token from the preceding form[cite: 1426].
[cite_start]
- **`_test_security_level_workflow`**: Submits a POST request to change the security level to "impossible," a value not available in the user interface, thus bypassing the standard selection workflow[cite: 1427].

[cite_start]
- **`_test_upload_workflow`**: Directly POSTs a file for upload without first visiting the upload page, which is the expected first step in the workflow[cite: 1428].

Step to Reproduce

[cite_start]
1. The script starts the workflow circumvention test[cite: 1425].
   [cite_start]
2. It focuses on the security level change workflow[cite: 1427].
3. The script crafts a POST request directly to `http://localhost:8080/security.php`. In this request, it sets the `security` parameter to `"impossible"`, a value that is not a standard option and would normally not be submitted.
4. The application accepts and processes this request without validating if "impossible" is a legitimate state.
   [cite_start]
5. The script observes a response indicating the change was successful and logs: "[✗] Successfully set security to impossible without proper workflow!"[cite: 1427], confirming the vulnerability.

Log Evidence

```
2025-08-05 02:50:28.577233 | [cite_start]----- Starting OTG-BUSLOGIC-006 (Circumvention of Work Flows) ----- [cite: 1425]
2025-08-05 02:50:28.577233 | [cite_start][i] Testing password change workflow circumvention [cite: 1426]
2025-08-05 02:50:28.632340 |   [cite_start][✓] Password change blocked without token [cite: 1426]
2025-08-05 02:50:28.632340 | [cite_start][i] Testing security level change workflow circumvention [cite: 1427]
2025-08-05 02:50:28.687536 |   [cite_start][✗] Successfully set security to impossible without proper workflow! [cite: 1427]
2025-08-05 02:50:28.687536 | [cite_start][i] Testing file upload workflow circumvention [cite: 1428]
```

```
2025-08-05 02:50:28.796433 |   [cite_start][✓] File upload
workflow enforced [cite: 1428]
```

Test Defenses Against Application Mis-use (OTG-BUSLOGIC-007)

Test Objective

The objective of this test is to determine if the application has protective measures to detect and respond to patterns of misuse. This includes checking for rate limiting against rapid requests, monitoring for multiple invalid inputs, detecting forced Browse to sensitive files, and identifying suspicious changes in user agent or location.

Target Endpoint

Various endpoints are tested to simulate different attack patterns:

[cite_start]
- Rate Limiting: `/vulnerabilities/brute/`, `/vulnerabilities/sqli/`, `/vulnerabilities/exec/` [cite: 1431]
[cite_start]
- Forced Browse: `/config.inc.php`, `/phpinfo.php`, `/.git/HEAD`, `/admin/` [cite: 1453, 1454]
- Session/UA tests: `/index.php`

Methodology

The `OTG_BUSLOGIC_007` function orchestrates several sub-tests to probe the application's defenses:

- **`_test_rate_limiting`**: Sends 10 rapid-fire POST requests with a malicious payload to see if an IP block or a "Too Many Requests" (429) response is triggered.
- **`_test_input_monitoring`**: Submits a sequence of different attack payloads (SQLi, XSS, LFI) to an input field to check if a Web Application Firewall (WAF) or other monitoring system blocks the requests after detecting a pattern of abuse.

- **`_test_session_termination`**: Performs several overtly suspicious actions (like UNION-based SQLi) and then checks if the user's session is still active or has been terminated.
- **`_test_forced_Browse`**: Attempts to directly access known sensitive files and directories.
- **`_test_geo_ua_detection`**: Changes the User-Agent header to a known hacking tool's signature (e.g., `sqlmap`) to see if the request is flagged or blocked.

Step to Reproduce

[cite_start]
1. The script begins by testing rate-limiting defenses[cite: 1431].
   [cite_start]
2. It sends 10 consecutive malicious requests to the brute-force page [cite: 1431-1436].
   [cite_start]All 10 requests receive a normal `200 OK` response, indicating no rate-limiting is in place[cite: 1436].
   [cite_start]
3. Next, it tests for forced Browse protection by requesting `http://localhost:8080/phpinfo.php`[cite: 1453]. The server responds with the `phpinfo` page, indicating the resource was not protected.
   [cite_start]
4. Finally, it tests for User-Agent-based blocking by changing the User-Agent to `sqlmap/1.6#stable` and making a request[cite: 1456]. The request is processed normally.
   [cite_start]
5. The script concludes that no misuse defenses for these scenarios were observed[cite: 1448, 1451, 1452, 1455, 1456].

Log Evidence

```
2025-08-05 02:55:39.249020 | [cite_start]----- Starting OTG-BUSLOGIC-007 (Defenses Against Application Mis-use) -----
[cite: 1430]
2025-08-05 02:55:39.252502 |
[cite_start][i] Testing rate limiting defenses [cite: 1431]
...
2025-08-05 02:55:40.759547 |   [cite_start][✗] No rate limiting
detected on http://localhost:8080/vulnerabilities/brute/ after
```

```
10 rapid requests [cite: 1436]
...
2025-08-05 02:55:46.309480 |
[i] Testing input monitoring defenses
2025-08-05 02:55:46.308605 |   [cite_start][✗] No input
monitoring detected after multiple malicious payloads [cite:
1451]
2025-08-05 02:55:46.309480 |
[i] Testing session termination defenses
2025-08-05 02:55:46.330432 |   [cite_start][✗] Session remained
active after suspicious activity [cite: 1452]
2025-08-05 02:55:46.330432 |
[i] Testing forced Browse detection
2025-08-05 02:55:46.343990 |   [cite_start][✗] Accessed
protected resource: http://localhost:8080/phpinfo.php [cite:
1453]
...
2025-08-05 02:55:46.347794 |   [cite_start][✗] No forced Browse
detection observed [cite: 1455]
2025-08-05 02:55:46.347794 |
[i] Testing geo/UA change detection
2025-08-05 02:55:46.353325 |   [cite_start][✗] No detection of
UA/geo changes [cite: 1456]
```

Test Upload of Unexpected File Types (OTG-BUSLOGIC-008)

Test Objective

This test assesses the file upload functionality's robustness by attempting to upload files with extensions that should be disallowed but are not necessarily malicious in content. The goal is to determine if the application relies solely on simple extension blacklisting/whitelisting and whether it can be bypassed. Examples include uploading `.html`, `.exe`, or `.htaccess` files.

Target Endpoint

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

Methodology

The `OTG_BUSLOGIC_008` function automates the testing of file type restrictions. The script defines a list of dictionaries, where each dictionary represents a file to be tested and contains its name, content, and MIME content type.

The script then iterates through this list, attempting to upload each file. For each attempt, it checks the server's response. A response containing "successfully uploaded" is flagged as a failure, while any other response is considered a successful rejection of the unexpected file type. The test also includes a specific check for path traversal vulnerabilities using a specially crafted ZIP file in the `_test_zip_path_traversal` method.

Step to Reproduce

[cite_start]
1. The script logs in and starts the unexpected file upload test[cite: 1457, 1458].
   [cite_start]
2. It attempts to upload a file named \`test.html\` with a content type of \`text/html\`[cite: 1458].
3. The server responds to the request, and the script analyzes the response body. The phrase "successfully uploaded" is not found.
   [cite_start]
4. The script correctly concludes that the upload was blocked and logs "[✓] File was rejected"[cite: 1459].
   [cite_start]
5. This process is repeated for other file types like \`.php\`, \`.jsp\`, \`.exe\`, and a malicious ZIP file, all of which are also rejected by the application's validation logic[cite: 1460, 1461, 1462, 1465].

Log Evidence

```
2025-08-05 03:00:55.669499 | [cite_start]----- Starting OTG-
BUSLOGIC-008 (Upload of Unexpected File Types) ----- [cite:
1458]
2025-08-05 03:00:55.669499 |
[cite_start][i] Testing upload of test.html (text/html) [cite:
```

```
1458]
2025-08-05 03:00:55.724459 |   [cite_start][✓] File was
rejected [cite: 1459]
2025-08-05 03:00:56.730042 |
[cite_start][i] Testing upload of test.jsp (text/plain) [cite:
1460]
2025-08-05 03:00:57.829501 |   [cite_start][✓] File was
rejected [cite: 1460]
2025-08-05 03:00:58.838059 |
[cite_start][i] Testing upload of test.exe (application/x-
msdownload) [cite: 1461]
2025-08-05 03:00:58.888104 |   [cite_start][✓] File was
rejected [cite: 1461]
...
2025-08-05 03:01:03.054533 | [cite_start][i] Testing ZIP file
with path traversal [cite: 1465]
2025-08-05 03:01:03.104765 |   [cite_start][✓] ZIP file was
rejected [cite: 1465]
```

Test Upload of Malicious Files (OTG-BUSLOGIC-009)

Test Objective

This test focuses on whether the application's file upload mechanism can be abused to upload actively malicious files, such as web shells, antivirus test files (EICAR), or files crafted to evade filters (e.g., using double extensions like `shell.php.jpg` or null byte characters). The goal is to determine if an attacker can place an executable file on the server or bypass content-based security checks.

Target Endpoint

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

Methodology

The `OTG_BUSLOGIC_009` function leverages a comprehensive list of malicious file types and filter evasion techniques. The script's methodology is as follows:

1. It defines a list of malicious test files, including various web shells (`.php`, `.phtml`), filter evasion names (`.php.jpg`, `.asp;.jpg`), the EICAR standard antivirus test file, and a denial-of-service XML payload ("Billion Laughs Attack").
2. It iterates through each malicious file, attempting to upload it to the server.
3. The `_check_malicious_upload` helper function analyzes the response. If an upload is successful, it goes a step further and attempts to access the uploaded file. For web shells, it tries to execute a simple command to confirm if the shell is active.
4. The script also includes a dedicated test for ZIP-based attacks like directory traversal and ZIP bombs using the `_test_zip_attacks` helper.

Step to Reproduce

[cite_start]
1. The script logs into DVWA and begins the malicious file upload test[cite: 1467].
   [cite_start]
2. It first attempts to upload a simple PHP web shell named `shell.php`[cite: 1468].
3. The server responds, and the script checks the response content. The response does not indicate a successful upload.
   [cite_start]
4. The script logs that the malicious file was correctly blocked: "[✓] File was rejected" [cite: 1468].
   [cite_start]
5. This process is repeated for numerous other malicious file types, including the EICAR test file, an XML bomb, and a ZIP bomb, all of which are successfully rejected by the application's security controls[cite: 1474, 1476, 1478].

Log Evidence

```
2025-08-05 03:04:57.901368 | [cite_start]----- Starting OTG-BUSLOGIC-009 (Upload of Malicious Files) ----- [cite: 1467]
2025-08-05 03:04:57.907802 |
 [cite_start][i] Testing upload of shell.php (application/x-php)
 [cite: 1468]
2025-08-05 03:04:57.958454 |   [cite_start][✓] File was rejected [cite: 1468]
```

```
2025-08-05 03:04:58.961647 |
[cite_start][i] Testing upload of eicar.txt (text/plain) [cite:
1474]
2025-08-05 03:05:04.280311 |   [cite_start][✓] File was
rejected [cite: 1474]
2025-08-05 03:05:05.284341 |
[cite_start][i] Testing upload of billion_laughs.xml
(application/xml) [cite: 1476]
2025-08-05 03:05:06.386058 |   [cite_start][✓] File was
rejected [cite: 1476]
2025-08-05 03:05:06.391445 |
[cite_start][i] Testing small ZIP bomb (safe test) [cite: 1478]
2025-08-05 03:05:07.498971 |   [cite_start][✓] ZIP bomb was
rejected [cite: 1478]
```