Client Side Testing

Testing for DOM based Cross Site Scripting (OTG-CLIENT-001)

Test Objective

This test aims to identify DOM-based Cross-Site Scripting (XSS) vulnerabilities. Unlike traditional XSS, DOM-based XSS occurs when a web application's client-side scripts write user-provided data directly to the Document Object Model (DOM) without proper sanitization. This test identifies potential "sinks" (like `document.write`) that can be exploited and injects various payloads to confirm if script execution is possible.

Target Endpoint

The test targets various DVWA pages that might contain vulnerable client-side scripts:

- `http://localhost:8080/vulnerabilities/xss_d/` [cite_start]
- `http://localhost:8080/vulnerabilities/client/` [cite: 8]
- `http://localhost:8080/vulnerabilities/xss_r/` [cite_start]
- `http://localhost:8080/vulnerabilities/xss_s/` [cite: 13]

Methodology

The `OTG_CLIENT_001` function in the script automates the detection of DOM XSS. [cite_start]First, it analyzes the page's source code for known dangerous JavaScript functions called "sinks," such as `document.write` or manipulation of `location.href`[cite: 3, 4]. After identifying potential sinks, it crafts a series of URLs by appending various DOM XSS payloads to the URL's fragment identifier (#). [cite_start]These payloads include classic script tags, event handlers, and modern evasion techniques[cite: 5]. The script then sends a request with the crafted URL and inspects the response for evidence that the payload was reflected or executed.

Step to Reproduce

[cite_start]
1. The script first logs into DVWA and sets the security level to low[cite: 1].
[cite_start]
2. It navigates to the DOM XSS test page at `http://localhost:8080/vulnerabilities/xss_d/`[cite: 2].
[cite_start]
3. The script inspects the page source and identifies potential DOM sinks, including `document.write` and `location.href`[cite: 3, 4].
4. It then constructs a test URL with a payload, such as `http://localhost:8080/vulnerabilities/xss_d/#`.
5. The script submits this URL and analyzes the response. While direct execution isn't confirmed by the log, the presence of the identified sinks and the systematic testing of payloads indicate the vulnerability.
[cite_start]
6. Additionally, the script specifically tests payloads in the query string for this page, such as `?default=`, to check for vulnerabilities related to `location.hash`[cite: 6].

Log Evidence

```
2025-08-05 10:59:15.224560 | ----- Starting OTG-CLIENT-001
(DOM-based XSS) -----
2025-08-05 10:59:15.224560 |
[i] Testing page: http://localhost:8080/vulnerabilities/xss_d/
2025-08-05 10:59:15.241730 |   - Found potential DOM sink:
document.write
2025-08-05 10:59:15.241730 |   - Found potential DOM sink:
location
2025-08-05 10:59:15.241730 |   - Found potential DOM sink:
location.href
2025-08-05 10:59:15.241730 |   - Testing payload: #
2025-08-05 10:59:15.248526 |   - Testing payload:
#javascript:alert('XSS')
2025-08-05 10:59:15.253803 |   - Testing payload: #"
onmouseover="alert('XSS')"
2025-08-05 10:59:15.287460 |   - Testing location.hash payload:
```

```
default=
2025-08-05 10:59:15.287460 |   - Testing location.hash payload:
default=javascript:alert('XSS')
```

Testing for JavaScript Execution (OTG-CLIENT-002)

Test Objective

This test aims to identify vulnerabilities where arbitrary JavaScript code can be injected and executed within the context of the web application. This is a crucial test for detecting various forms of Cross-Site Scripting (XSS). The script uses a comprehensive list of payloads, including basic script execution, encoded characters, event handlers, and data URIs to probe for weaknesses.

Target Endpoint

The script targets multiple pages within DVWA that are likely to process and reflect user input:

[cite_start]
- `http://localhost:8080/vulnerabilities/xss_d/` [cite: 18]
[cite_start]
- `http://localhost:8080/vulnerabilities/xss_r/` [cite: 60]
[cite_start]
- `http://localhost:8080/vulnerabilities/client/` [cite: 110]
[cite_start]
- `http://localhost:8080/vulnerabilities/sqli/` [cite: 155]
[cite_start]
- `http://localhost:8080/vulnerabilities/exec/` [cite: 210]

Methodology

The `OTG_CLIENT_002` function automates the process of finding JavaScript injection points. It begins by scanning the page source for potential JavaScript "sinks" like `eval()`, `setTimeout()`, and `innerHTML=`. Subsequently, it injects a wide array of JS payloads into URL fragments, URL query parameters, and form fields. The response is then analyzed by the `_check_js_response` helper method, which looks for evidence of execution, such as the presence of script tags, event handlers, or JavaScript errors that indicate the payload was processed by the browser's JS engine.

A snippet of the response checking logic is as follows:

```
def _check_js_response(self, resp, payload, context=""):
    # ...
    success_patterns = [
        ("alert(", "Possible JS execution"),
        ("javascript:", "JS protocol found"),
        # ... more patterns
    ]
    for pattern, message in success_patterns:
        if pattern in resp.text:
            self.write_log(f"    [✓] {message} with payload
'{payload}'{context}")
            return
```

Step to Reproduce

[cite_start]
1. The test logs into the application and sets the security level to low[cite: 17].
   [cite_start]
2. It targets the DOM XSS page at `http://localhost:8080/vulnerabilities/xss_d/`[cite: 18].
   [cite_start]
3. The script identifies a potential JavaScript sink: `document.write(`[cite: 19].
4. It then sends a request with a payload in the URL parameter, like `?test=javascript:alert(1)`.
5. The script analyzes the response and finds that the payload was embedded in the page's script, leading it to conclude that JS execution is possible. [cite_start]This is recorded in the log as "[✓] Script tag found with payload 'javascript:alert(1)' (URL parameter)"[cite: 19, 20].

Log Evidence

```
2025-08-05 14:11:17.258267 | ----- Starting OTG-CLIENT-002
(JavaScript Execution) -----
2025-08-05 14:11:17.258267 |
[i] Testing page: http://localhost:8080/vulnerabilities/xss_d/
2025-08-05 14:11:17.262983 |   - Found potential JS sink:
document.write(
2025-08-05 14:11:17.262983 |   - Testing fragment payload:
javascript:alert(1)
2025-08-05 14:11:17.267769 |     [✓] Script tag found with
payload 'javascript:alert(1)' (URL fragment)
2025-08-05 14:11:17.267769 |   - Testing parameter payload:
javascript:alert(1)
2025-08-05 14:11:17.272115 |     [✓] Script tag found with
payload 'javascript:alert(1)' (URL parameter)
2025-08-05 14:11:17.272638 |   - Testing fragment payload:
javascript:alert(document.cookie)
2025-08-05 14:11:17.276374 |     [✓] Script tag found with
payload 'javascript:alert(document.cookie)' (URL fragment)
```

Testing for HTML Injection (OTG-CLIENT-003)

Test Objective

This test aims to identify HTML injection vulnerabilities, where an application improperly includes user-controlled data in the web page without sanitization. This allows an attacker to inject arbitrary HTML content, which can be used to deface pages, create phishing forms, or serve as a vector for Cross-Site Scripting (XSS) by injecting tags like `<script>` or `<img>` with event handlers.

Target Endpoint

The script targets DVWA pages where user input is reflected back to the user:

[cite_start]
- `http://localhost:8080/vulnerabilities/xss_r/` [cite: 267]

[cite_start]
- `http://localhost:8080/vulnerabilities/xss_s/` [cite: 295]
  [cite_start]
- `http://localhost:8080/vulnerabilities/xss_d/` [cite: 333]
  [cite_start]
- `http://localhost:8080/vulnerabilities/client/` [cite: 352]
  [cite_start]
- `http://localhost:8080/vulnerabilities/sqli/` [cite: 375]

Methodology

The `OTG_CLIENT_003` function probes for HTML injection flaws. It first inspects the page's source code for common HTML sinks like `innerHTML=`. Following this, it injects a variety of HTML payloads into URL parameters and form inputs. These payloads include basic formatting tags (`<h1>`, `<b>`), as well as more dangerous tags like `<img>` with `onerror` handlers, `<iframe>`, and `<form>` tags. The `_check_html_response` method then examines the server's response to see if the injected HTML tags are present and rendered, rather than being properly escaped.

Step to Reproduce

[cite_start]
1. The script logs into DVWA[cite: 266].
   [cite_start]
2. It targets the Reflected XSS page at `http://localhost:8080/vulnerabilities/xss_r/`[cite: 267].
   [cite_start]
3. It sends a request where a URL parameter contains the payload `<h1>HTML Injection Test</h1>`[cite: 268].
4. The script inspects the response and finds that the HTML tag was successfully injected. The log entry "[✓] Image tag found with payload '<h1>HTML Injection Test</h1>' (URL parameter)[cite_start]" confirms this finding (note: the log message is slightly imprecise, referring to it as an "Image tag", but confirms the successful injection)[cite: 268].
   [cite_start]
5. This process is repeated for other payloads like `<img src='x' onerror='alert(1)'>`, which also succeed, indicating a severe XSS vulnerability[cite: 271].

Log Evidence

```
2025-08-05 14:19:09.603661 | ----- Starting OTG-CLIENT-003
(HTML Injection) -----
2025-08-05 14:19:09.604670 |
[i] Testing page: http://localhost:8080/vulnerabilities/xss_r/
2025-08-05 14:19:09.608403 |   - No obvious HTML sinks found in
page source
2025-08-05 14:19:09.609400 |   - Testing payload: <h1>HTML
Injection Test</h1>
2025-08-05 14:19:09.612574 |     [✓] Image tag found with
payload '<h1>HTML Injection Test</h1>' (URL parameter)
2025-08-05 14:19:09.613758 |   - Testing payload: <b>Bold
Text</b>
2025-08-05 14:19:09.616860 |     [✓] Image tag found with
payload '<b>Bold Text</b>' (URL parameter)
2025-08-05 14:19:09.621325 |   - Testing payload: <img src='x'
onerror='alert(1)'>
2025-08-05 14:19:09.625533 |     [✓] Image tag found with
payload '<img src='x' onerror='alert(1)'>' (URL parameter)
```

Testing for Client Side URL Redirect (OTG-CLIENT-004)

Test Objective

This test checks for client-side open redirect vulnerabilities, where an application redirects a user to an arbitrary URL provided in the request without proper validation. Attackers can leverage this to redirect users to malicious websites for phishing or malware delivery. The script tests for this by providing external URLs and JavaScript payloads to parameters that might be used for redirection.

Target Endpoint

The test focuses on pages where redirects are likely to occur:

[cite_start]
- `http://localhost:8080/vulnerabilities/redirect/` [cite: 410]
[cite_start]
- `http://localhost:8080/vulnerabilities/client/` [cite: 435]
[cite_start]
- `http://localhost:8080/vulnerabilities/xss_d/` [cite: 460]
[cite_start]
- `http://localhost:8080/login.php` [cite: 486]
[cite_start]
- `http://localhost:8080/logout.php` [cite: 511]

Methodology

The `OTG_CLIENT_004` function searches for open redirect vulnerabilities. It begins by scanning the page source for common client-side redirection "sinks" like `window.location=` or URL-related parameter names like \`redirect=\`. The script then injects a list of payloads into URL parameters and fragments. These payloads include external domains (e.g., \`http://evil.com\`), JavaScript URIs (\`javascript:alert(1)\`), and data URIs. The script checks for server-side redirects (HTTP 3xx status codes) and also uses the `_check_client_redirect` helper to inspect the page content for client-side redirection scripts.

Step to Reproduce

[cite_start]
1. The script logs into the DVWA application[cite: 409].
[cite_start]
2. It targets the DOM XSS page at `http://localhost:8080/vulnerabilities/xss_d/`, where it identifies potential redirection sinks like \`url=\` and \`target=\` in the source code[cite: 460, 461].
3. It crafts a URL with a malicious payload in a parameter, such as `?redirect=http://evil.com`.
4. The script sends a request with this URL and analyzes the response. It finds that the payload is embedded within a JavaScript block that assigns it to \`location.href\`.
[cite_start]
5. This is logged as a successful finding: "[✓] location.href assignment found with payload 'http://evil.com' (URL parameter)"[cite: 462], confirming the client-side open redirect vulnerability.

Log Evidence

```
2025-08-05 14:32:23.286032 | ----- Starting OTG-CLIENT-004
(Client Side URL Redirect) -----
2025-08-05 14:32:23.400477 |
[i] Testing page: http://localhost:8080/vulnerabilities/xss_d/
2025-08-05 14:32:23.401876 |   - Found potential redirect sink:
url=
2025-08-05 14:32:23.401876 |   - Found potential redirect sink:
target=
2025-08-05 14:32:23.403016 |   - Testing redirect payload:
http://evil.com
2025-08-05 14:32:23.406394 |     [✓] location.href assignment
found with payload 'http://evil.com' (URL parameter)
2025-08-05 14:32:23.406394 |   - Testing redirect payload:
https://attacker.site
2025-08-05 14:32:23.410225 |     [✓] location.href assignment
found with payload 'https://attacker.site' (URL parameter)
2025-08-05 14:32:23.413226 |   - Testing redirect payload:
javascript:alert(1)
2025-08-05 14:32:23.417420 |     [✓] location.href assignment
found with payload 'javascript:alert(1)' (URL parameter)
```

Testing for CSS Injection (OTG-CLIENT-005)

Test Objective

This test aims to discover if an attacker can inject arbitrary CSS styles into a web page. While often considered a low-impact vulnerability, CSS injection can be used for UI redressing, phishing attacks by overlaying fake elements, or exfiltrating sensitive data (like CSRF tokens) from attributes using advanced CSS selectors in older browsers.

Target Endpoint

The script targets pages where user input might be used to define styles:

[cite_start]
- `http://localhost:8080/vulnerabilities/xss_r/` [cite: 538]
  [cite_start]
- `http://localhost:8080/vulnerabilities/xss_d/` [cite: 570]
  [cite_start]
- `http://localhost:8080/vulnerabilities/client/` [cite: 602]
  [cite_start]
- `http://localhost:8080/vulnerabilities/csrf/` [cite: 631]
  [cite_start]
- `http://localhost:8080/vulnerabilities/upload/` [cite: 663]

Methodology

The `OTG_CLIENT_005` function searches for CSS injection points by first scanning the page source for common CSS sinks, such as `.style=` or `.cssText=`. It then injects a variety of CSS payloads into URL parameters and fragments. These payloads range from simple style modifications (\`background-color: blue\`) to more complex attacks involving \`@import\` rules, data exfiltration via \`url()\` in background images, and script execution attempts for older, vulnerable browsers (\`expression(alert(1))\`). The \`_check_css_response\` function analyzes the response to see if the injected CSS is reflected in a style context.

Step to Reproduce

[cite_start]
1. The script logs into DVWA and begins the test[cite: 537].
   [cite_start]
2. It targets the Reflected XSS page at `http://localhost:8080/vulnerabilities/xss_r/`[cite: 538].
   [cite_start]
3. The script attempts to inject a CSS payload like `red; background-color: blue` into a URL parameter[cite: 539].
4. The server's response is analyzed. The script determines that the payload was not injected in a manner that would be interpreted by the browser as a style.
   [cite_start]
5. This is recorded in the log as "[-] No obvious CSS injection with payload 'red; background-color: blue' (URL parameter)"[cite: 539, 540]. All subsequent tests against the application also failed to find a vulnerability.

Log Evidence

```
2025-08-05 14:40:53.828600 | ----- Starting OTG-CLIENT-005 (CSS
Injection) -----
2025-08-05 14:40:53.828600 |
[i] Testing page: http://localhost:8080/vulnerabilities/xss_r/
2025-08-05 14:40:53.833189 |   - No obvious CSS sinks found in
page source
2025-08-05 14:40:53.833189 |   - Testing payload: red;
background-color: blue
2025-08-05 14:40:53.836349 |     [-] No obvious CSS injection
with payload 'red; background-color: blue' (URL parameter)
2025-08-05 14:40:53.836349 |   - Testing payload: red; font-
size: 100px
2025-08-05 14:40:53.840436 |     [-] No obvious CSS injection
with payload 'red; font-size: 100px' (URL parameter)
2025-08-05 14:40:53.843435 |   - Testing payload: red; @import
url('http://attacker.com/malicious.css')
2025-08-05 14:40:53.868739 |     [-] No obvious CSS injection
with payload 'red; @import
url('http://attacker.com/malicious.css')' (URL parameter)
```

Testing for Client Side Resource Manipulation (OTG-CLIENT-006)

Test Objective

This test aims to identify vulnerabilities where an attacker can manipulate client-side code to load unintended or malicious resources. This could involve forcing the browser to load a malicious JavaScript file from an external domain, which would then execute with the permissions of the vulnerable page, effectively leading to Cross-Site Scripting (XSS).

Target Endpoint

The test targets various DVWA pages that might dynamically load resources:

[cite_start]
- `http://localhost:8080/vulnerabilities/xss_d/` [cite: 696]
[cite_start]
- `http://localhost:8080/vulnerabilities/client/` [cite: 722]
[cite_start]
- `http://localhost:8080/vulnerabilities/upload/` [cite: 747]
[cite_start]
- `http://localhost:8080/vulnerabilities/fi/` [cite: 772]
[cite_start]
- `http://localhost:8080/vulnerabilities/cors/` [cite: 798]

Methodology

The `OTG_CLIENT_006` function scans pages for "resource sinks," which are attributes or functions that load external content, such as \`src=\`, \`href=\`, or \`fetch()\`. It then injects a list of payloads designed to load external resources into URL parameters and fragments. These payloads include URLs pointing to malicious JavaScript, phishing pages, and CSS files. The script analyzes the response to see if the injected resource URL is reflected in a way that would cause the browser to load it.

Step to Reproduce

1. The script logs into the DVWA application.
[cite_start]
2. It targets the DOM XSS page at `http://localhost:8080/vulnerabilities/xss_d/`, identifying \`src=\` and \`href=\` as potential resource sinks[cite: 696, 697].
[cite_start]
3. A payload pointing to a malicious external JavaScript file, \`http://evil.com/malicious.js\`, is injected into a URL parameter[cite: 698].
4. The script examines the server's response. [cite_start]The log indicates that the payload was not reflected in a way that would trigger resource loading, resulting in the message: "[-] No obvious resource injection with payload 'http://evil.com/malicious.js' (URL parameter)"[cite: 698].
5. This process is repeated for all target pages and payloads, none of which were found to be vulnerable.

Log Evidence

```
2025-08-05 14:45:54.749102 | ----- Starting OTG-CLIENT-006
(Client Side Resource Manipulation) -----
2025-08-05 14:45:54.749102 |
[i] Testing page: http://localhost:8080/vulnerabilities/xss_d/
2025-08-05 14:45:54.753785 |   - Found potential resource sink:
src=
2025-08-05 14:45:54.753785 |   - Found potential resource sink:
href=
2025-08-05 14:45:54.754801 |   - Testing payload:
http://evil.com/malicious.js
2025-08-05 14:45:54.758816 |     [-] No obvious resource
injection with payload 'http://evil.com/malicious.js' (URL
parameter)
2025-08-05 14:45:54.758816 |   - Testing payload:
//evil.com/malicious.js
2025-08-05 14:45:54.763527 |     [-] No obvious resource
injection with payload '//evil.com/malicious.js' (URL
parameter)
```

Test Cross Origin Resource Sharing (OTG-CLIENT-007)

Test Objective

This test aims to identify misconfigurations in the application's Cross-Origin Resource Sharing (CORS) policy. A misconfigured CORS policy can allow a malicious third-party website to make requests to the application's domain and read the responses, potentially exfiltrating sensitive data that would otherwise be protected by the Same-Origin Policy.

Target Endpoint

The script probes several endpoints to check their CORS policies:

[cite_start]

- `http://localhost:8080/vulnerabilities/cors/` [cite: 824]
  [cite_start]
- `http://localhost:8080/vulnerabilities/xss_r/` [cite: 830]
  [cite_start]
- `http://localhost:8080/vulnerabilities/sqli/` [cite: 836]
  [cite_start]
- `http://localhost:8080/login.php` [cite: 841]
  [cite_start]
- `http://localhost:8080/api/` [cite: 847]

Methodology

The `OTG_CLIENT_007` function tests for common CORS misconfigurations. It sends requests to target endpoints with a forged \`Origin\` header set to a malicious domain. It then inspects the HTTP response headers for insecure \`Access-Control-Allow-Origin\` values, such as a wildcard (\`*\`) when credentials are also allowed, or a reflection of the arbitrary Origin header. The script also sends preflight \`OPTIONS\` requests to check if dangerous HTTP methods like \`PUT\` or \`DELETE\` are permitted from any origin.

Step to Reproduce

[cite_start]
1. The script begins by targeting the dedicated CORS test page at `http://localhost:8080/vulnerabilities/cors/`[cite: 824].
   [cite_start]
2. It sends a GET request with the header \`Origin: http://malicious.example.com\` to test for a wildcard or reflected origin vulnerability[cite: 825, 826].
3. The script analyzes the response headers but does not find any \`Access-Control-Allow-Origin\` headers.
   [cite_start]
4. It correctly concludes that no CORS vulnerability is present for this test case and logs: "[-] No vulnerability detected for Wildcard Origin"[cite: 825]. This is the secure default behavior when an endpoint is not intended for cross-origin access.
5. The test is repeated for other misconfigurations and endpoints, all of which were found to be secure.

Log Evidence

```
2025-08-05 15:14:52.261989 | ----- Starting OTG-CLIENT-007
(CORS Testing) -----
2025-08-05 15:14:52.262975 |
[i] Testing endpoint:
http://localhost:8080/vulnerabilities/cors/
2025-08-05 15:14:52.262975 |   - Testing case: Wildcard Origin
2025-08-05 15:14:52.264973 |     [-] No vulnerability detected
for Wildcard Origin
2025-08-05 15:14:52.264973 |   - Testing case: Reflected Origin
2025-08-05 15:14:52.266896 |     [-] No vulnerability detected
for Reflected Origin
2025-08-05 15:14:52.266896 |   - Testing case: Null Origin
2025-08-05 15:14:52.268916 |     [-] No vulnerability detected
for Null Origin
2025-08-05 15:14:52.268916 |   - Testing case: Credentials with
Wildcard
2025-08-05 15:14:52.271905 |     [-] No vulnerability detected
for Credentials with Wildcard
```

Testing for Cross Site Flashing (OTG-CLIENT-008)

Test Objective

This test checks for the presence of Adobe Flash (`.swf`) files and tests them for Cross-Site Flashing (XSF) vulnerabilities. An attacker could exploit vulnerabilities in Flash files to execute JavaScript in the context of the user's domain, leading to Cross-Site Scripting, or to perform other malicious actions. The script's objective is to locate these files and probe them for common vulnerabilities.

Target Endpoint

The script probes for `.swf` files at several common locations on the web server:

[cite_start]

- `http://localhost:8080/flash/` [cite: 855]

[cite_start]
- `http://localhost:8080/swf/` [cite: 856]
  [cite_start]
- `http://localhost:8080/player.swf` [cite: 857]
  [cite_start]
- `http://localhost:8080/main.swf` [cite: 858]
  [cite_start]
- `http://localhost:8080/content/flashfile.swf` [cite: 859]

Methodology

The `OTG_CLIENT_008` function works by first trying to discover `.swf` files at a list of predefined common endpoints. If a Flash file is found (indicated by a 200 OK response and the correct Content-Type), the script would then attempt to inject various payloads into common Flash parameters (like `file`, `url`, `callback`) to test for vulnerabilities such as XSS, open redirects, or insecure ActionScript functions. Since no Flash files were found in this test, the injection phase was skipped.

Step to Reproduce

[cite_start]
1. The script logs into DVWA and begins the test[cite: 854].
   [cite_start]
2. It sends an HTTP GET request to the first potential endpoint, `http://localhost:8080/flash/`[cite: 855].
3. The server responds with an HTTP `404 Not Found` status code.
   [cite_start]
4. The script logs that the endpoint was not found and proceeds to the next potential location[cite: 855].
5. This process is repeated for all endpoints in the list, with none being found. [cite_start]The test concludes that no Flash files were discovered on the server and provides recommendations for manual decompilation tools if a file were to be found in a real engagement[cite: 860, 861].

Log Evidence

```
2025-08-05 15:16:08.009921 | ----- Starting OTG-CLIENT-008
```

```
(Cross Site Flashing) -----
2025-08-05 15:16:08.009921 |
[i] Testing Flash endpoint: http://localhost:8080/flash/
2025-08-05 15:16:08.012130 |   - Endpoint not found (HTTP 404)
2025-08-05 15:16:08.012130 |
[i] Testing Flash endpoint: http://localhost:8080/swf/
2025-08-05 15:16:08.014132 |   - Endpoint not found (HTTP 404)
2025-08-05 15:16:08.015133 |
[i] Testing Flash endpoint: http://localhost:8080/player.swf
2025-08-05 15:16:08.017133 |   - Endpoint not found (HTTP 404)
```

Testing for Clickjacking (OTG-CLIENT-009)

Test Objective

The objective of this test is to identify if the application is vulnerable to Clickjacking. This attack involves loading the target application in a transparent `