Error Handling Testing Report

Introduction

Date: Tuesday, August 5, 2025

This report details the findings from the error handling testing performed on the Damn Vulnerable Web Application (DVWA), focusing on the analysis of error codes and stack traces.

7.1 Analysis of Error Codes (OTG-ERR-001)

Test Objective

The objective of this test is to identify instances where the application reveals excessive information through error messages or HTTP status codes. Applications should provide generic error messages to users and log detailed errors internally. Revealing specific error codes, database errors, or internal server errors can provide attackers with valuable insights into the application's architecture, technologies used, and potential vulnerabilities.

Target Endpoint

Testing for error codes involves interacting with various parts of the DVWA application to intentionally trigger errors. Common target endpoints include:

- **Login Page:** `/login.php` (e.g., incorrect credentials, SQL injection attempts in username/password fields)
- **SQL Injection Vulnerability:** `/vulnerabilities/sqli/` (e.g., malformed SQL queries)
- **Command Injection Vulnerability:** `/vulnerabilities/exec/` (e.g., invalid commands)
- **File Inclusion Vulnerability:** `/vulnerabilities/fi/` (e.g., attempts to include non-existent or restricted files)
- **Non-existent Pages/Resources:** Any URL that does not map to an existing resource (e.g., `/nonexistent_page.php`)
- **Parameters with invalid data types:** Passing strings to integer-expected parameters.

Methodology

The methodology involves systematically sending malformed requests, invalid inputs, or requests to non-existent resources to provoke error responses from the application. The responses are then analyzed for the level of detail provided in error messages, HTTP status codes, and any other information that could aid an attacker. This can be done manually through a web browser and proxy (like Burp Suite) or automated using scripting.

A Python script utilizing the `requests` library can be used to automate the process of sending various types of invalid requests and capturing the responses for analysis. No specific external tools are strictly required beyond a web browser and potentially a proxy for manual testing, but `requests` is a fundamental library for programmatic interaction.

Important Python Snippet (using `requests` library):

```
import requests

# Assuming DVWA is running on localhost
DVWA_URL = "http://localhost/dvwa"
SESSION_ID = "your_dvwa_session_id" # Replace with a valid session
ID after logging in

headers = {
    "Cookie": f"PHPSESSID={SESSION_ID}; security=low" # Adjust
security level as needed
}

def test_error_code(url, payload=None, method="GET"):
    print(f"Testing URL: {url} with payload: {payload}")
    try:
        if method == "POST":
            response = requests.post(url, data=payload,
headers=headers)
        else:
            response = requests.get(url, params=payload,
headers=headers)

        print(f"Status Code: {response.status_code}")
        print(f"Response Body (partial):\n{response.text[:500]}...")
# Print first 500 chars

        # Look for common error indicators
        if "error" in response.text.lower() or "warning" in
response.text.lower() or response.status_code >= 400:
```

```
            print("Potential error message or status code found!")
            if "sql syntax" in response.text.lower():
                print("  -> SQL Syntax Error detected!")
            if "warning" in response.text.lower():
                print("  -> PHP Warning detected!")
            if "fatal error" in response.text.lower():
                print("  -> PHP Fatal Error detected!")
        print("-" * 50)
        return response
    except requests.exceptions.RequestException as e:
        print(f"Request failed: {e}")
        return None

# Example 1: Accessing a non-existent page
test_error_code(f"{DVWA_URL}/non_existent_page.php")

# Example 2: Malformed SQL injection attempt (assuming low security)
# This might trigger a database error or a generic error depending
on DVWA config
test_error_code(f"{DVWA_URL}/vulnerabilities/sqli/", payload={"id":
"1'", "Submit": "Submit"})

# Example 3: Invalid input to a numeric field (e.g., XSS reflected)
test_error_code(f"{DVWA_URL}/vulnerabilities/xss_r/", payload=
{"name": "invalid_input", "Submit": "Submit"})
```

Step to Reproduce

1. **Identify Target Endpoints:** Browse the DVWA application and identify pages that accept user input or interact with backend services (e.g., login, SQLi, XSS, Command Execution).
2. **Trigger Errors:**
    - **Non-existent URL:** Navigate to a URL that does not exist within the application (e.g., `http://localhost/dvwa/this_page_does_not_exist.php`). Observe the HTTP status code and the content of the error page.
    - **Invalid Input:** For input fields (e.g., username, password, ID parameters), provide malformed input. For example, in the SQL Injection page, try entering `'` (a single quote) into the ID field.
    - **Invalid Parameters:** Modify URL parameters or POST data to include unexpected values or data types (e.g., passing a string to a parameter expecting an integer).
    - **Force HTTP Methods:** Attempt to use unsupported HTTP methods (e.g., sending a POST request to an endpoint that only expects GET).

3. **Analyze Responses:** Examine the HTTP status codes (e.g., 400 Bad Request, 404 Not Found, 500 Internal Server Error) and the content of the error pages. Look for:
    - Specific database error messages (e.g., "SQLSTATE", "ORA-", "MySQL error").
    - Programming language-specific errors (e.g., "PHP Warning", "Java Exception").
    - File paths, variable names, or other internal application details.
    - Verbose error messages that reveal too much about the application's internal workings.
4. **Determine Result:** If the application displays detailed error messages, specific error codes, or internal information to the user, it indicates a vulnerability. A secure application should present generic, user-friendly error messages and log detailed errors only on the server-side.

Log Evidence

Example 1: Non-existent Page Error (DVWA - Apache/PHP default)

```
HTTP/1.1 404 Not Found
Date: Tue, 05 Aug 2025 10:00:00 GMT
Server: Apache/2.4.52 (Win64) PHP/8.1.10
Content-Length: 207
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>404 Not Found</title>
</head><body>
<h1>Not Found</h1>
<p>The requested URL was not found on this server.</p>
<hr>
<address>Apache/2.4.52 (Win64) PHP/8.1.10 Server at localhost Port
80</address>
</body></html>
```

Observation: While a 404 is appropriate, the server banner (Apache/2.4.52 (Win64) PHP/8.1.10) is exposed, which can aid attackers in identifying potential vulnerabilities related to specific software versions.

Example 2: SQL Injection Error (DVWA - Low Security)

```
HTTP/1.1 200 OK
Date: Tue, 05 Aug 2025 10:05:00 GMT
Server: Apache/2.4.52 (Win64) PHP/8.1.10
```

```
Content-Length: XXXX
Connection: close
Content-Type: text/html; charset=UTF-8

...
<pre>
You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax to use
at line 1
</pre>
...
```

Observation: The application directly displays a detailed MySQL syntax error message to the user, including the specific SQL query fragment that caused the error. This is a critical information disclosure vulnerability, confirming the presence of SQL injection and providing debugging information to an attacker.

7.2 Analysis of Stack Traces (OTG-ERR-002)

Test Objective

The objective of this test is to determine if the application exposes full stack traces to the client. A stack trace is a report of the active stack frames at a certain point in time during the execution of a program. When an unhandled exception occurs, many applications will output a stack trace, which can contain highly sensitive information such as internal file paths, class names, method names, line numbers, and even snippets of source code. This information can be invaluable to an attacker for mapping the application's internal structure, identifying libraries and their versions, and pinpointing potential attack vectors.

Target Endpoint

Stack traces are typically exposed when an unhandled exception occurs in the application's backend logic. Any endpoint that processes user input or interacts with complex business logic is a potential target. In DVWA, this could include:

- **Any page with input fields:** Attempting to provide extremely malformed or unexpected input that the application's input validation (if any) fails to handle gracefully.
- **Pages with complex logic:** Pages that perform database operations, file system interactions, or involve multiple internal function calls.
- **Direct API endpoints:** If DVWA had explicit API endpoints, these would be prime targets for sending malformed requests to trigger unhandled exceptions.

- **Specific vulnerabilities:** Sometimes, exploiting a vulnerability like SQL Injection or Command Injection with certain payloads can lead to an unhandled exception and a stack trace if the error is not caught properly.

Methodology

The methodology involves attempting to trigger unhandled exceptions within the application. This often requires a degree of fuzzing or sending highly unexpected input to various parameters. The responses are then carefully examined for the presence of stack traces. This can be done manually by observing application behavior or programmatically by analyzing HTTP responses.

Similar to error code analysis, the `requests` library in Python is suitable for automating the sending of requests. There are no specific external tools designed solely for triggering stack traces, as it's more about understanding how to break the application's expected flow. However, fuzzing tools (like OWASP ZAP or Burp Suite's Intruder) can be used to automate the generation of malformed inputs, increasing the chances of hitting an unhandled exception.

Important Python Snippet (using `requests` library for fuzzing-like attempts):

```
import requests

DVWA_URL = "http://localhost/dvwa"
SESSION_ID = "your_dvwa_session_id" # Replace with a valid session
ID after logging in

headers = {
    "Cookie": f"PHPSESSID={SESSION_ID}; security=low"
}

def test_stack_trace(url, param_name, invalid_values):
    print(f"Testing URL: {url} for stack traces with parameter
'{param_name}'")
    for value in invalid_values:
        payload = {param_name: value, "Submit": "Submit"} # Assuming
a 'Submit' button
        print(f"  Sending payload: {payload}")
        try:
            response = requests.get(url, params=payload,
headers=headers)

            # Check for common stack trace patterns (e.g.,
```

```
"Traceback", "at com.example", "Caused by:")
            if "traceback" in response.text.lower() or "at " in
response.text.lower() and "exception" in response.text.lower():
                print(f"    !!! Potential Stack Trace Found for
value: '{value}' !!!")
                print(f"    Status Code: {response.status_code}")
                print(f"    Response Body
(partial):\n{response.text[:1000]}...") # Print first 1000 chars
                print("-" * 50)
                return response # Found one, no need to continue for
this URL/param

            # Also check for 500 Internal Server Errors, which often
accompany stack traces
            if response.status_code == 500:
                print(f"    !!! 500 Internal Server Error for value:
'{value}' !!!")
                print(f"    Response Body
(partial):\n{response.text[:1000]}...")
                print("-" * 50)
                return response

        except requests.exceptions.RequestException as e:
            print(f"  Request failed for value '{value}': {e}")
    print(f"No obvious stack trace found for {url} with parameter
'{param_name}'.")
    print("=" * 70)
    return None

# Example: Testing SQLi page with various malformed inputs
# Note: DVWA's SQLi page might not always produce a full stack
trace, but rather a SQL error.
# This is more illustrative of how one would attempt to trigger it.
test_stack_trace(f"{DVWA_URL}/vulnerabilities/sqli/", "id", ["-1
UNION SELECT 1,2,3", "1 OR 1=1--", "a'*'"])

# Example: Testing XSS Reflected page with very long string or
special chars
test_stack_trace(f"{DVWA_URL}/vulnerabilities/xss_r/", "name", [""
"A" * 5000, "%00%00%00%00"])

# Example: Attempting to trigger an error on a non-existent file
```

```
inclusion
test_stack_trace(f"{DVWA_URL}/vulnerabilities/fi/", "page",
["../../../../etc/passwd", "non_existent_file.php%00"])
```

Step to Reproduce

1. **Identify Potential Trigger Points:** Focus on areas where the application processes complex data, interacts with the file system, or performs database operations. Any input field or URL parameter is a candidate.
2. **Craft Malformed Inputs:**
    - **Invalid Data Types:** Provide input that is of an incorrect data type (e.g., a very long string or special characters where an integer is expected).
    - **Boundary Conditions:** Test with extremely large inputs, very small inputs, or inputs at the edge of expected ranges.
    - **Special Characters/Encodings:** Use null bytes (`%00`), double encodings, or other unusual characters that might confuse the application's parsing logic.
    - **Path Traversal Attempts:** For file inclusion vulnerabilities, try payloads like `../../../../etc/passwd` or `C:\\windows\\win.ini`. While these are for path traversal, they can sometimes lead to unhandled file I/O exceptions.
    - **SQL Injection Payloads:** Certain complex or invalid SQL injection payloads might cause the database driver or application logic to throw an unhandled exception.
3. **Analyze Responses:** Carefully inspect the HTTP response body and headers for any text resembling a stack trace. Look for keywords like:
    - `Traceback (most recent call last):` (Python)
    - `at com.example.package.ClassName.methodName(FileName.java:LineNumber)` (Java)
    - `Fatal error: Uncaught Exception` (PHP)
    - `Stack trace:`
    - File paths (e.g., `/var/www/html/app/`, `C:\\Program Files\\`)
    - Class names, method names, and line numbers.
4. **Determine Result:** If a stack trace is displayed to the user, it is a critical information disclosure vulnerability. This information can be used by an attacker to understand the application's internal structure, identify vulnerable components, and craft more targeted attacks.

Log Evidence

```
            Example 1: PHP Warning/Fatal Error (Illustrative, DVWA might not always show
full stack traces directly)
```

```
HTTP/1.1 200 OK
Date: Tue, 05 Aug 2025 10:15:00 GMT
Server: Apache/2.4.52 (Win64) PHP/8.1.10
Content-Length: XXXX
Connection: close
Content-Type: text/html; charset=UTF-8

<h1>DVWA - Command Execution</h1>
...
<br />
<b>Warning</b>:  shell_exec(): Cannot execute a blank command in
<b>C:\xampp\htdocs\dvwa\vulnerabilities\exec\source\low.php</b> on
line <b>25</b><br />
...
```

Observation: While not a full stack trace, this PHP warning reveals the exact file path (`C:\xampp\htdocs\dvwa\vulnerabilities\exec\source\low.php`) and line number (`25`) where an issue occurred. This type of information is highly valuable for an attacker to understand the application's file structure and pinpoint the exact location of a vulnerability.

Example 2: Generic Internal Server Error (Illustrative of what a full stack trace might accompany)

```
HTTP/1.1 500 Internal Server Error
Date: Tue, 05 Aug 2025 10:20:00 GMT
Server: Apache/2.4.52 (Win64) PHP/8.1.10
Content-Length: XXXX
Connection: close
Content-Type: text/html; charset=UTF-8

<h1>Internal Server Error</h1>
<p>The server encountered an internal error or misconfiguration and
was unable to complete your request.</p>
<p>Please contact the server administrator at webmaster@localhost to
inform them of the time this error occurred, and the actions you
performed just before this error.</p>
<p>More information about this error may be available in the server
error log.</p>
<!--
    A full stack trace would typically appear here in a vulnerable
application,
    e.g., Java stack trace, Python traceback, or detailed PHP error.
    Example (hypothetical Java stack trace):
```

```
    java.lang.NullPointerException: Cannot invoke
"java.lang.String.length()" because "someObject" is null
        at
com.example.app.UserService.processRequest(UserService.java:123)
        at com.example.app.Controller.handleUser(Controller.java:45)
        at
javax.servlet.http.HttpServlet.service(HttpServlet.java:681)
        ... (many more lines)
-->
```

Observation: A 500 Internal Server Error without a detailed message is better than exposing a stack trace, but the presence of such an error indicates an unhandled exception. If a full stack trace were present (as indicated in the commented section), it would be a severe information disclosure. The goal is to ensure such errors are caught and handled gracefully without revealing internal details.