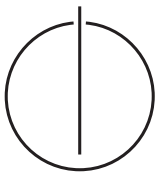
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Secure Coding Phase 5

Team 3: Aurel Roci, Stefan Kohler



Executive Summary

We managed to fix the most of the crucial vulnerabilities of the web application.

Contents

1 Time Tracking Table

Name	Task	Time
Aurel Roci	Test File Extensions Handling for Sensitive Information and documenting	1
	Test HTTP Methods and documenting	1
	Test HTTP Strict Transport Security and documenting	1
	Test RIA cross domain policy	1
	Error Handling	0.5
	Testing for default credentials	0.5
	Testing for Reflected Cross Site Scripting and documenting	0.5
	Testing for Stored Cross Site Scripting and documenting	0.5
	Testing for HTTP Verb Tampering and documenting	0.5
	Testing for SQL Injection and documenting	2
	Test Number of Times a Function Can be Used Limits	0.5
	Test Business Logic Data Validation and documenting	1
	Testing for Bypassing Session Management Schema and documenting	1
	Testing for Cookies attributes and documenting	1
	Testing for Session Fixation and documenting	1
	Testing for Exposed Session Variables and documenting	1
	Testing for Cross Site Request Forgery and documenting	1
	Testing for logout functionality and documenting	1
	Test Session Timeout and documenting	1
	Testing for Session puzzling and documenting	1
	Testing Report	2
	Testing for Cross Site Request Forgery	0.5
	Testing for Privilege Escalation	1
	Presentation	0.25
Stefan Ch. Kofler	Reverse-Engineer the binary file	10
	Binary-equivalent	9
	Decompile jar file	1

2 Application Architecture

Our applications consists of multiple files, most of them represent an own page in the web application, others are files that are included in this pages that perform some actions (e.g. creating a new user etc) and two models for the transactions and the users.

2.1 Models

We start by showing an UML class diagram of our two models.

2.1.1 Transaction

An *Transaction* object represents a single transaction. It contains all the necessary attributes that are also in the database table for the transactions. When fetching transaction from the database, these transactions are then mapped to *Transaction* objects.

The class has also two helper methods for checking if an *Transaction* is approved and to map it to JSON string.

Additionally the class has multiple class methods for fetching transactions, approving them, validating and checking TAN numbers, creating new transactions and to create a transaction that represents the initialization of a new user account with an specific balance.

2.1.2 User

User objects represent an user account. It stores all the attributes of an user like their name, email address etc. This model gets also mapped from the database when fetching existing users.

This class some helper methods for detecting if the user is an employee, if the account is approved, to get the account number in the right format, to get the current balance of the user and to get his last used TAN number.

The *User* class also has some class methods for fetching users, approving them and checking if the login system is blocked for this user.

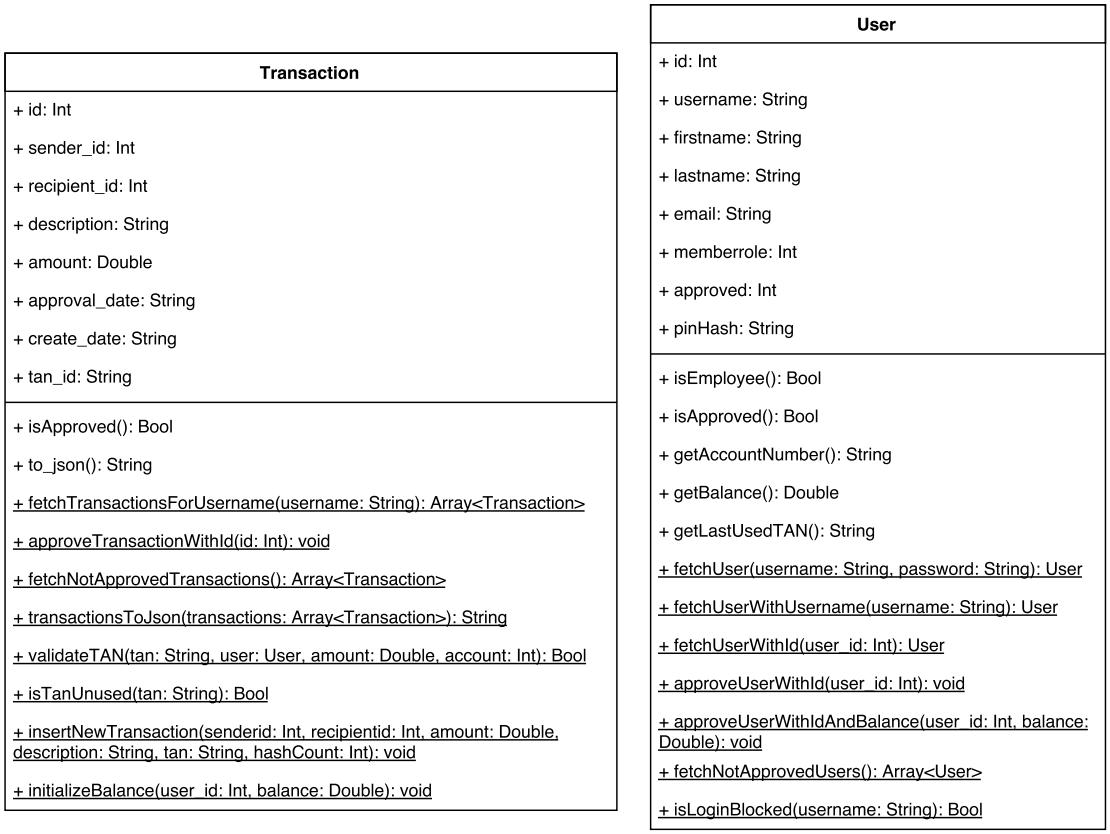


Figure 2.1: UML class diagram

2.2 Pages

Figure 2.2 shows an overview of all the different accessible web pages and the flow between them.

It starts by the user accessing any page of the web application. If the user isn't logged in already he gets redirected to the login page otherwise to the homepage of his user role.

In the login page the user can log himself in, register a new account or reset his password. For the last two methods he gets redirected to an specific webpage for registering/resetting the password.

After logging in the system detects if the user account is the one of an customer or an employee and loads the corresponding PHP file. They have different functionality but both of the include the same *transactions.php* file for showing the transaction overview for a specific user.

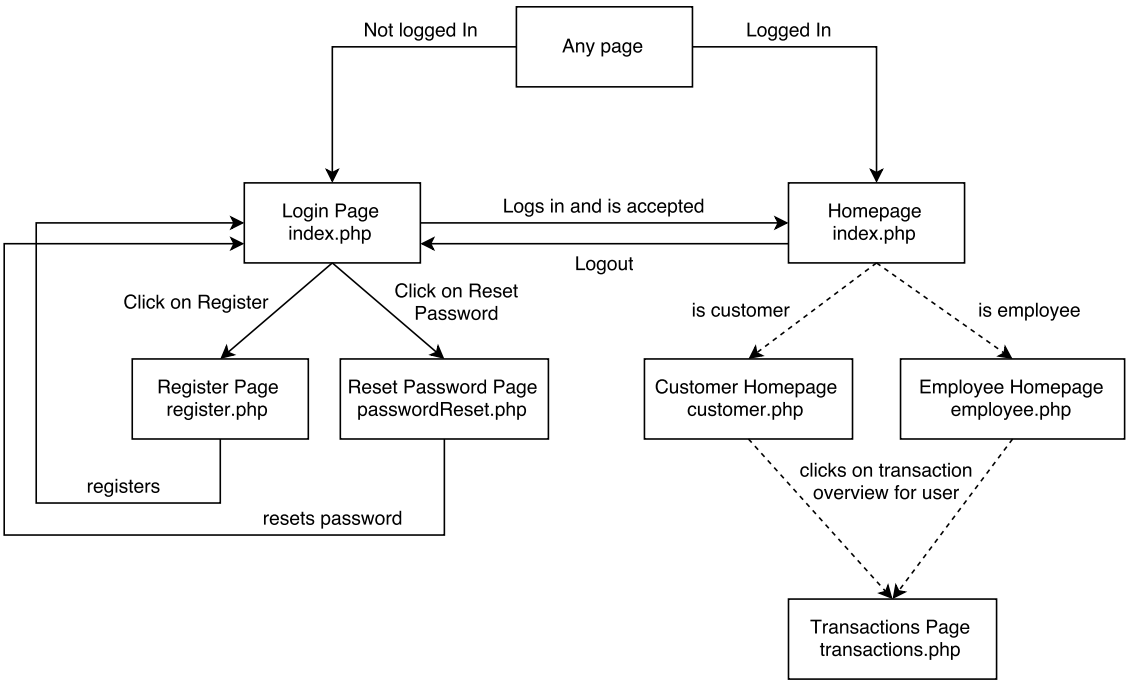


Figure 2.2: Overview of different accessible pages

The functionality of these pages is mostly in different php files with the suffix .inc. These files are included in the pages PHP file and perform the necessary tasks a user clicked on.

The *core.inc.php* file is included in all the web pages and provides some additional functionality needed in the web pages.

The *init.sec.php* file is there to check on every page request if the user is logged in and prevents clickhijacking on the client side by adding some javascript to the web pages.

2.3 Frameworks

The web application uses multiple external frameworks for providing functionality that would have been to much work to reimplement them on our own.

2.3.1 Bootstrap

Bootstrap is a very popular HTML, CSS, and JS framework for developing user interfaces on the web. It helped us to build a great looking web application without spending to much time on the User Interface but rather on functionality and security. The framework is available for free under the MIT license.

2.3.2 jspdf

We used *jspdf* to deliver pdf's to the client for their transaction history. It is an easy to use client-side javascript framework that builds the pdf files on the fly without having them to be stored on the web server. That makes our web application more space efficient.

2.3.3 fpdf

For the TAN's that had to be sent as an password protected pdf over email we used the *fpdf* framework. This framework is an pure PHP framework and not an javascript framework such as jspdf. It is also really easy to use and provides a lot of functionality. We used a second pdf framework because the requirements changed for phase 3 and jspdf wouldn't even have the option to create password protected PDF files.

2.3.4 PHPMailer

PHPMailer is the framework of our choice for sending emails over PHP. It is full-featured and allowed us to send emails and even add attachments to the emails (for the generated TAN pdf files).

2.4 Interaction with other systems

Our web service interacts with just one other system and that is the *Gravatar* service for getting a profile picture for the user without storing it on the web server.

Gravatar could be a potential security risk because the server isn't in our hands and if *Gravatar* would be hacked, they could find out our users email addresses or could send instead of the profile picture other pictures that could lead to misuse of the online banking application.

It should be mentioned that it is a very small security risk and that is not comparable to sites using *Google Analytics* and similar tools and some online banking solutions even use them. *Gravatar* a useful service and is completely free.

2.5 Hardware/Software mapping

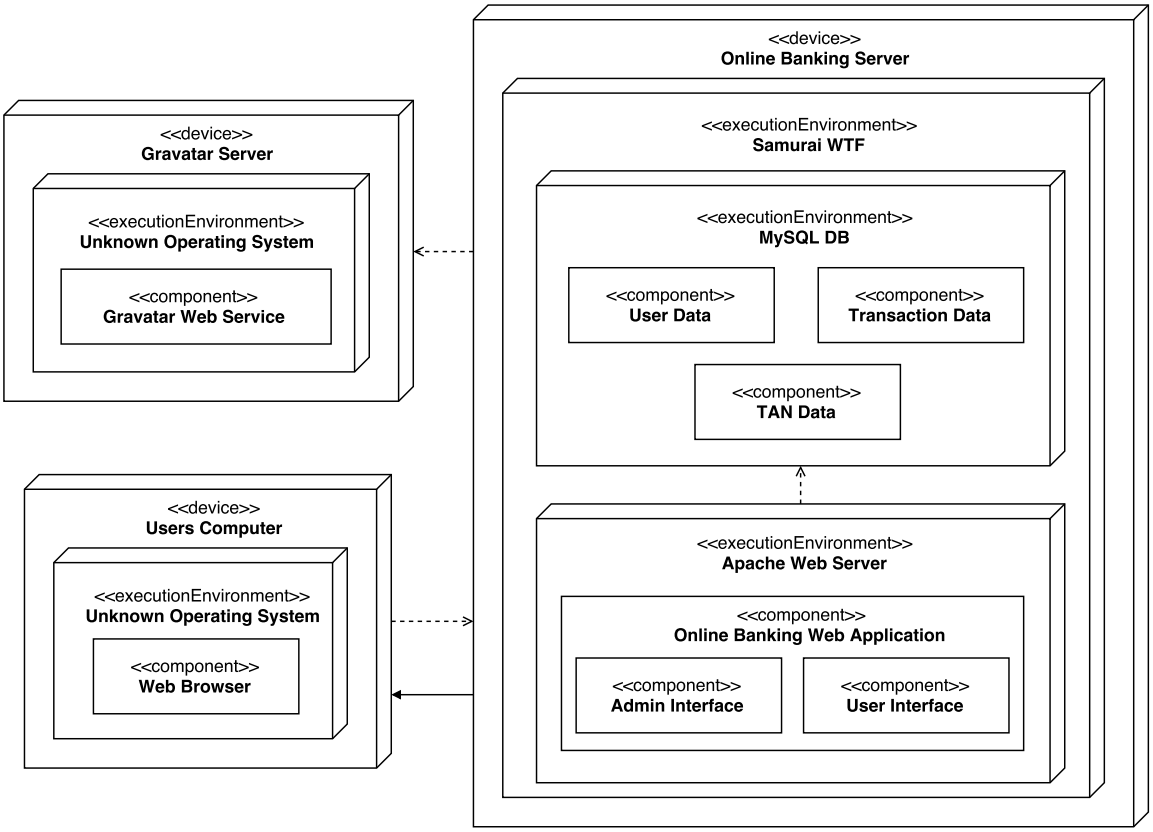


Figure 2.3: Hardware/Software Mapping (UML deployment diagram)

Our online banking web application runs on an device called *Online Banking Server* that is on our case a virtualized machine. This machine runs *Samurai WTF* as an operation system. On this system we have two major services running. First the *MySQL DB* database that contains the *User Data*, *Transaction Data*, and *TAN Data*. On the other hand we have our *Apache Web Server* on which our *Online Banking Web Application* written in PHP is running. The web application has two major components: The *Admin Interface* and the *User Interface*.

3 Security Measures

4 Fixes

4.0.1 Testing for Weak lock out mechanism(OTG-AUTHN-003)

After 3 wrong attempts to login the account now is blocked and an employee has now to approve it, for the user to be able to login.
online_banking/models/user.php lines 177-178:

```
$query = "UPDATE users SET approved = 0 WHERE
               username='".mysql_real_escape_string($username)."';";
$result = mysql_query($query);
```

After the user tries to log 3 times and fails the account will be blocked until an employee approves it again. So if the attacker deletes the session and tries again the account is blocked.

4.0.2 Testing Directory traversal/file include (OTG-AUTHZ-001)

Fixed by fixing Command Injection and Local File Inclusion

4.0.3 Test Session Timeout(OTG-SESS-007)

Added a functionality in *online_banking/init.sec.php* in lines 17-21 :

```
(isset($_SESSION['LAST_ACTIVITY']) && (time() - $_SESSION['LAST_ACTIVITY'] > 900)) {
session_unset();
session_destroy();
}
```

This checks the last activity done in the website. If there is no activity for 15 minutes then the session is destroyed.

4.0.4 Testing for Reflected Cross Site Scripting(OTG-INPVAL-001)

Insert a code in line 77 of the *online_banking/employee.php* :

```
$search = htmlspecialchars($search);
```

and line 54 of *online_banking/passwordReset.php*:

```
htmlspecialchars($_GET['id'])
```

to sanitize the input, so no code can be executed.

4.0.5 Testing for Stored Cross Site Scripting(OTG-INPVAL-002)

Insert code in lines 17 of *online_banking/customer.inc.php* to sanitized the input from XSS with the following code:

```
$description = htmlspecialchars($description);
```

Changed the code in line 22 of *online_banking/register.inc.php* from :

```
if(preg_match("/^$passwordRegex/", $password))
to:
if(preg_match("/^$emailRegex/", $email))
```

There was an if statement that checked for the password written twice, and we changed one to check for the email.

4.0.6 Testing for SQL Injection (OTG-INPVAL-005) and Mysql testing (OTG-INPVAL-005)

Modified the query in line 131 of the *online_banking/models/user.php* file to sanitize the input:

```
$query = "UPDATE users SET approved = 1 WHERE
                    id='".mysql_real_escape_string($user_id)."';";
```

Also we sanitized the input in lines 137-140:

```
$user_id = mysql_real_escape_string($user_id);
$balance = mysql_real_escape_string($user_id);
$user_id = htmlspecialchars($user_id);
$balance = htmlspecialchars($balance);
```

This should prevent *SQL Injections*.

4.0.7 Testing for Code Injection, Testing for Local File Inclusion, Testing for Remote File Inclusion(OTG-INPVAL-012)

Inserted a function when we read the variable in *online_banking/applicationDownload.php* line 2 :

```
$user_id = intval($_GET['user_id']);
```

to receive only integer values. This will prevent any code injections.

4.0.8 Testing for Command Injection(OTG-INPVAL-013)

Inserted code in line 3 of the *online_banking/applicationDownload.php* :

```
$user_id = escapeshellarg($user_id);
```

This way the shell commands are removed from the variable

4.0.9 Testing for incubated vulnerabilities(OTG-INPVAL-015)

This vulnerability was fixed by fixing XSS, SQL Injection, Code Injection, Command Injection.

4.0.10 Analysis of Error Codes (OTG-ERR-001)

Changed the PHP *error_reporting()* such that it does not display any errors: *online_banking/init.sec.php*; *online_banking/customer.sec.php*

4.0.11 Test Business Logic Data Validation(OTG-BUSLOGIC-001)

Added a server side check using regular expression in *online_banking/customer.sec.php* line 21:

```
if($amount<0 || preg_match("/^$amountRegex/", $amount))
```

line 8:

```
$amountRegex = "^\d*\.?\d*$";
```

This way the check for the amount of money to transfer is server-side too. This will prevent any input of incorrect amount, even if the client-side check is bypassed.

4.0.12 Test Integrity Checks(OTG-BUSLOGIC-003)

Fixed by fixing Injection vulnerabilities.

4.0.13 Testing for JavaScript Execution(OTG-CLIENT-002)

Fixed by fixing Injection vulnerabilities.

4.0.14 Testing for HTML Injection(OTG-CLIENT-003)

Fixed by fixing Injection vulnerabilities.

4.0.15 Testing for CSS Injection(OTG-CLIENT-005)

Fixed by fixing Injection vulnerabilities.