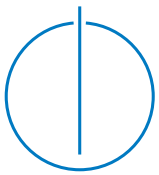
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Secure Coding - Phase 5

Final Report

Team 12
Alexander Lill
Lorenzo Donini
Florian Mauracher
Mahmoud Naser



Executive Summary

Not entirely sure what we should be writing down here...

Contents

1 Timetracking

Alexander Lill

Task	Time
Cleanup phase 5	1h
Setup of presentation and deliverables	1h
Defenses Against Application Mis-use - Fixing	2h
Defenses Against Application Mis-use - Report	1h
Business Logic Data Validation - Fixing	2h
Business Logic Data Validation - Report	1h
Documentation of database architecture	2h
Preparing the presentation	2h
Sum	0 h

Lorenzo Donini

Task	Time
Setting up report	1 h
Group meeting	1 h
Refactoring error codes	1 h
Fixing circumvention of workflows	0.5 h
Adapting error messages on transaction pages	1 h
Adapting error messages on client & employee pages	1 h
Report - business logic fixes	1 h
Report - error codes fixes	0.5 h
Report - security measures	1 h
Report - architecture overview	0.5 h
Report - SCS UML	0.5 h
Report - SCS architecture	1 h
Presentation - lessons learned	1 h
Project bugfixes	1 h
Sum	12 h

Florian Mauracher

Task	Time
Read report of Team 4	0.5 h
Discuss and coordinate fixes for Phase 5	0.5 h
Secure webserver SSL configuration	1 h
Reorganize files to restrict acess to project folders	1 h
Improve error messages for batch transactions	0.5 h
Document installation process	1 h
Sum	0 h

Mahmoud Naser

Task	Time
Sum	0 h

2 Application Architecture

The Goliath National Bank application was built using a PHP backend and a web-based (HTML + Javascript) frontend. Also, the solution may provide the clients of Goliath National Bank with an additional SCS (short for SmartCardSimulator) software, which is needed to provide 2-step authentication during transactions and can be used on any system that supports Java 1.7+.
The application was almost entirely developed without the aid of external libraries, preferring the use of builtin APIs and a custom architecture. Here is a list of the used libraries, which will be later on described more in details, along with their interaction with the system:

- PHPMailer (see `https://github.com/PHPMailer/PHPMailer`)
- fpdf (see `http://www.fpdf.org/`)
- pdf encryption script (see `http://www.fpdf.org/en/script/script37.php`)
- secureimage (see `https://github.com/dapphp/securimage`)

In the following section we will discuss the architecture of the whole web application; we will also provide dedicated sections for a more detailed analysis of the Entity-Relationship model used in the database as well as the architecture of the Smart Card Simulator.

2.1 Architecture

The architecture is based on an Apache web server, running the Goliath National Bank website, and a MySQL database, both hosted on the same machine and directly communicating with each other. The web server contains the whole business logic of the application and communicates with the client's Browser via HTTP, where the GUI is shown. No frameworks are used on the backend or on the frontend, since the web application was developed from scratch.
We will now analyze the design of the solution in more detail, paying attention both to the backend and the frontend.
The design is based on a simple model and multiple views, each of which is made up

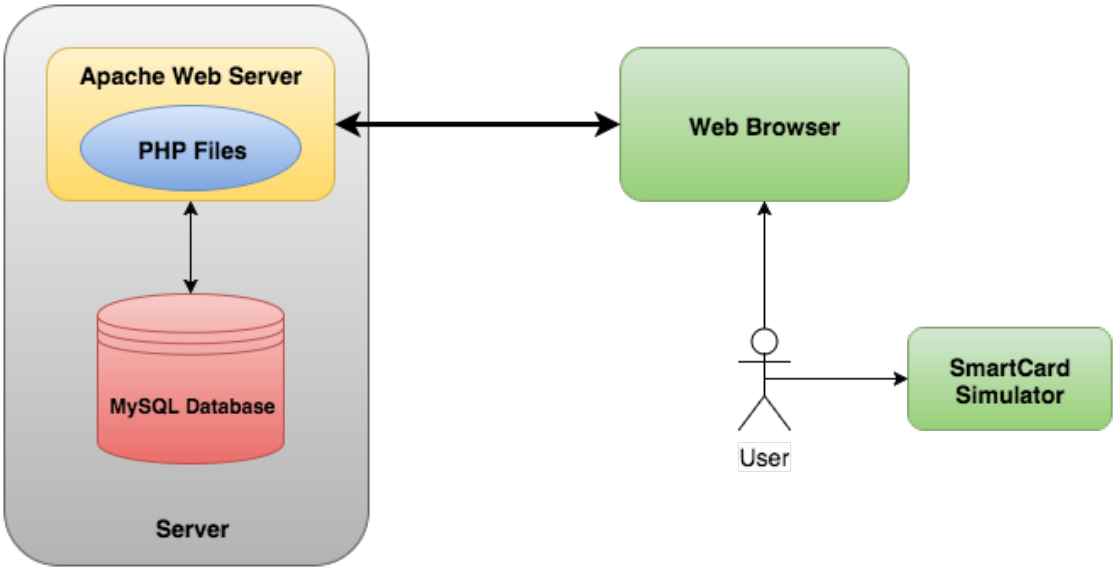


Figure 2.1: Architecture of the solution

of a controller for that specific view and the view itself, which is generated dynamically in some cases. The idea behind the whole design is to split different parts of each resulting webpage into logical subviews, making them independent from each other and allowing us to reuse them for different purposes.
All resources are divided into packages, according to their scope and category.

2.1.1 Resource Mapper

In order to provide a more fine-grained granularity, a `resource_mappings.php` component was implemented, allowing to query all resources via a single API. This file contains the mappings to all resources which views or pages may need to include: pages, classes, media and so on. By simply querying a logical name, a resource is returned, regardless of the package it is contained in. This way, different pages can be included at runtime, without having to resort to static paths. Also, no direct paths are ever used by other PHP classes, but always queried on `resource_mappings` using logical names, thus avoiding inclusion attacks.

```
require_once getpageabsolute("db_functions");
require_once getPageAbsolute("user");
require_once getpageabsolute("util");
```

Figure 2.2: Example of resource inclusion inside `new_transaction.php`

2.1.2 Model

The model used for the Goliath National Bank solution is made up of three main classes (see Figure 2.3):

- the **user**, in which all data relevant to the user is stored, i.e. ID, email, name, pin, preferred banking method and so on. This class also allows to approve/reject/block a user directly, using the related methods;
- each user can be associated to N accounts, each of which is identified by a unique ID and contains the current balance of the user as well as a reference to all past transactions;
- a **transaction** contains all the details concerning a specific transaction, including the ID, the source and destination account and so on.

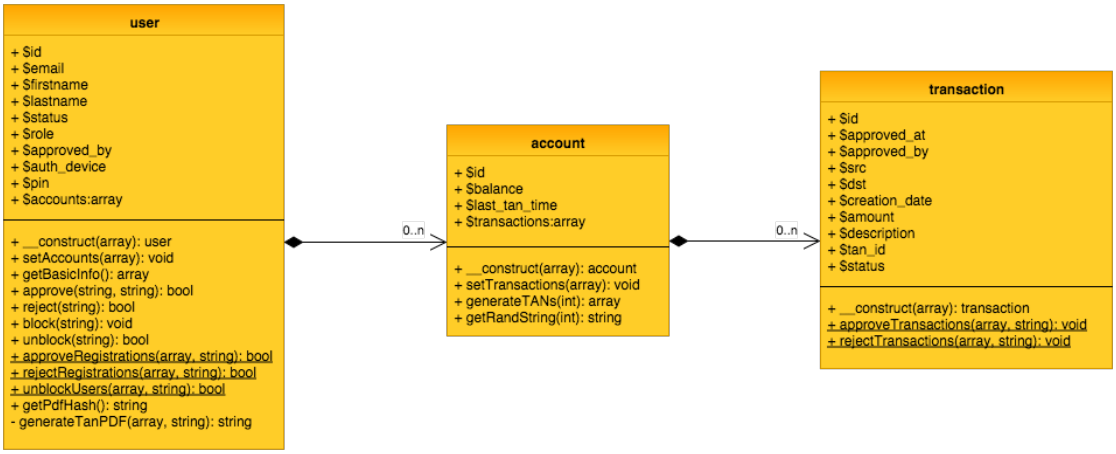


Figure 2.3: Model UML

All model objects reflect the data stored on the database: after a successful query, the array containing the data read from the database is passed to the constructor of the an object, which will populate the fields of that object with the values passed in the array.

2.1.3 Views

The application tries to reuse as many view components as possible, by building pages dynamically as shown in **??**. While the `index.php`, `login.php` and `registration.php` pages follow an ad-hoc design, once logged in, all users refer directly to either the `employee.php` or the `client.php` page, depending on their role: whatever the operation,

the requested page will always be one of these two; any direct request to another page will either redirect the user to the homepage or to a 404 error page. Both employees and clients have a similar GUI, with different sections (and site functionalities) that can be directly chosen from the navigation bar on top.
Depending on the parameters passed by the client inside a POST request, the server will handle the request in order to generate the appropriate content for the page, which is achieved by simply including the correct pages (i.e. resources). The main parameters include:

- a **section**, which is chosen by the user via the navigation bar. There are only a few possible sections, each one generating a different GUI. The server queries the resource mapper with a logical name for the requested section, then includes it inside the container view (being either the employee or the client page);
- a **frame**, which is chosen by the user via a menu found on the left isde of each section. The server queries the resource mapper with a logical name for the requested frame, then includes it inside the container view, in this case being the section.

Using this "hierarchical" inclusion mechanism, we can generate complex pages by keeping the structure of the respective container views intact. All PHP pages that provide HTML output perform strict access controls to avoid malicious attackers from accessing arbitrary pages or circumventing the application work-flow.

2.2 Database Schema

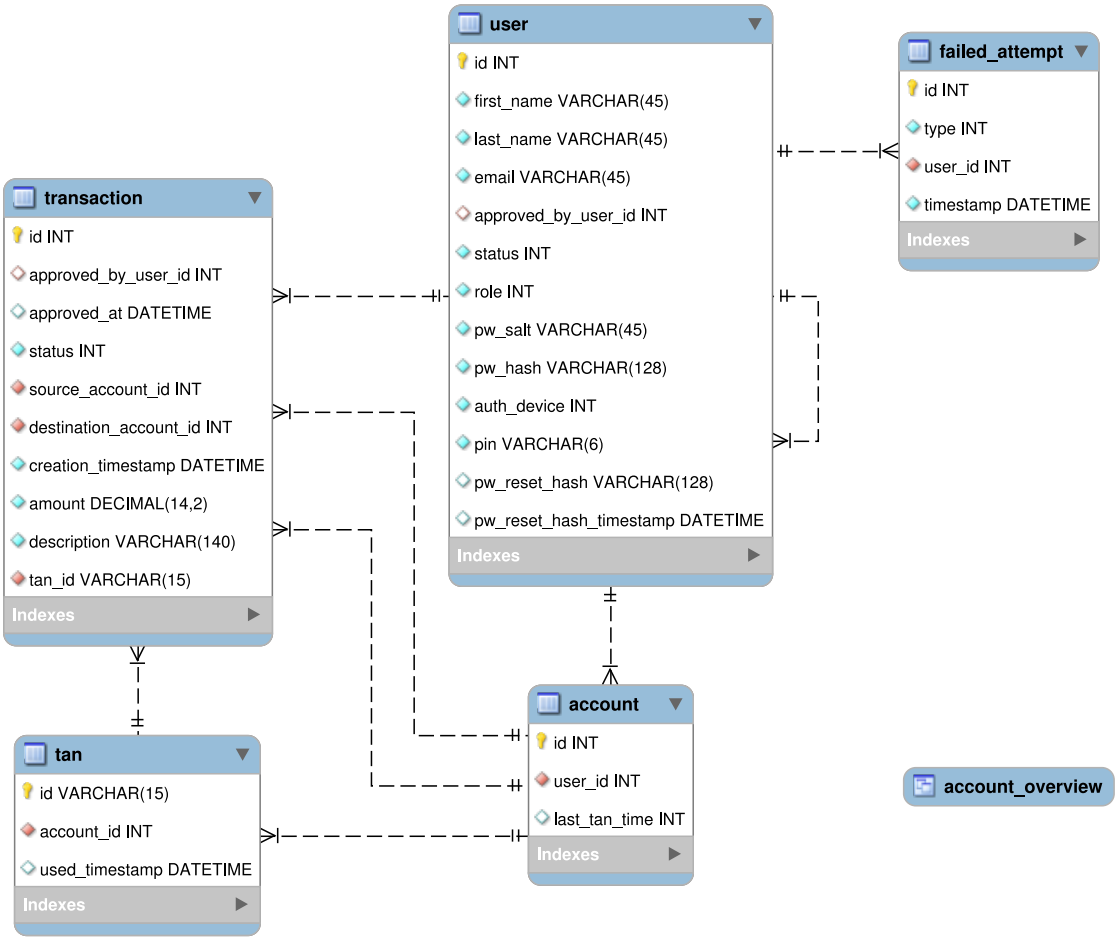


Figure 2.4: Database model

The database schema was created using the *MySQL Workbench*. The database schema is shown in Figure 2.4 and describes the following 6 entities:

Table "user"
Contains all the user attributes including user ID, full name, mail, status (`unapproved`, `approved`, `rejected`, `blocked`), role (`client`, `employee`), salt and hash for the authentication, the device used for TAN authentication (`none` for users without account, otherwise `TANs` or the `SCS`), the PIN as well as the password reset hash and creation timestamp for this hash. Additionally the user ID is stored which approved/blocked/rejected the user the last time.

Table "account"
Contains the account number, the user ID of the owner and the timestamp of the last TAN that was used if the user is using the SCS (see section 2.3).

Table "transaction"
Contains information about the status (`unapproved`, `approved`, `rejected`), the user ID of the approving/rejecting user and approval/rejection time of the transactions as well as the source and destination account, creation timestamp, amount and description of the transaction as well as the used TAN (if the SCS was not used).

Table "tan"
Contains the TANs and which account they belong to as well as a field specifying if the given TAN has been used (value is a timestamp) or not used (value is `NULL`).

Table "failed_attempt"
Contains all failed attempts (e.g. failed `login` or invalid `tan`) and their timestamp associated to the user ID that attempted the action.

View "account_overview"
Is used to obtain the balance for a given account number. As shown in Figure 2.5 this approach is not trivial. First (lines 8-9) we set Barney's balance to a fixed amount because all the welcome credits to new users are coming from his account. Then we exclude some transactions from our calculation, namely rejected transactions, unapproved transactions and transactions that have the same source- and destination account number (see lines 12-14). After that we determine the sign of the transaction — it is either positive if the given account is the destination account, or negative if our account is the source account (see lines 17-19).

```sql
CREATE OR REPLACE VIEW `account_overview` AS

SELECT
    A.id,
    A.user_id,
    A.last_tan_time,
    IF (
        A.user_id = 1,  # if barney
        1000000000,     # balance is 1 billion
            SUM(        # else calculate sum
                IF (
                    (T.status = 2)                                      # if rejected transaction
                    OR (T.status = 0 AND A.id = T.destination_account_id)   # or unapproved for destination
                    OR (T.source_account_id = T.destination_account_id),    # or destination equals source
                    0,                                                  # then do not use for balance
                    IF (                                                # otherwise check
                        A.id = T.destination_account_id,                # if we are destination
                        T.amount,                                       # then count as plus
                        -T.amount                                       # otherwise count as minus
                    )
                )
            )
    ) AS balance
FROM
    account A LEFT JOIN transaction T
    ON T.destination_account_id = A.id OR T.source_account_id = A.id
GROUP BY A.id;
```

Figure 2.5: View "account_overview"

2.3 Smart Card Simulator

The SCS is a stand-alone Java application that can be downloaded as a `.jar` file via the web application without having to be logged in as a user, i.e. the SCS is free for anyone, although only clients of the bank can make use of it. This is because the SCS allows users to generate TANs on the fly when performing transactions, instead of reading pre-generated TANs from a finite list.
The Goliath National Bank Smart Card Simulator does not require any kind of direct interaction with the PHP backend when used, hence clients can even run it from a machine disconnected from the internet and/or different than the one they are performing the transaction from. Also, the executable file is not bound to a specific account and could, therefore, be used by multiple clients as well (or for multiple accounts).

2.3.1 Usage

When using the SCS, users are required to insert their personal PIN and all details regarding a specific transaction (either manually or contained in a batch file). The SCS will then generate a pseudo-random TAN based on the user's input and a timestamp; the web application will challenge the client to input the same data inside the transaction HTML form as well, validating it against the generated TAN. In case the TAN is proved to be invalid, the transaction fails; this is either due to a user not inserting the same values inside the SCS and the transaction page, or the user providing an invalid PIN inside the application.

2.3.2 Class Design

The solution includes the minimal amount of features required for the functionality to work. By doing so, we kept a clean and simple design, based on an MVP pattern (the architecture can be seen in Figure 2.7). We will now briefly describe the solution and the involved components.
• The **MainView** class captures user actions callbacks and dispatches them to the Presenter class, which acts as the controller of the whole application. The view was created via a GUI Builder, using Java Swing graphical components.
Event listeners are created inline inside the `initComponents` private method and associated to some objects, like the `generateButton`. When choosing to generate a TAN, the `performGenerateTan` method is automatically called within the MainView, which will retrieve the user's input and call the appropriate method inside the presenter

Figure 2.6: Graphical User Interface of the SmartCard Simulator

component, depending whether the user wants to perform a batch transaction or a single transaction.
Upon a successful operation, the generated TAN will be displayed by the MainView, otherwise an error will be shown to the user.
• The **Presenter** is in charge of the business logic. When a TAN generation request is dispatched to the presenter, this component checks the validity of all user input (or the contents of the batch file) and prepares it for being processed by the CustomTanGenerator class. After CustomTanGenerator has processed the data and generated a TAN, the presenter will take care of returning it to the MainView. Since the application does not involve a real model, the only data which can be retrieved is given by the user's input.
• **CustomTanGenerator** is accessed statically by the presenter, as it contains only the algorithm used to generate a TAN given some input parameters. This class does not need to keep any state and will directly return the generated TAN.

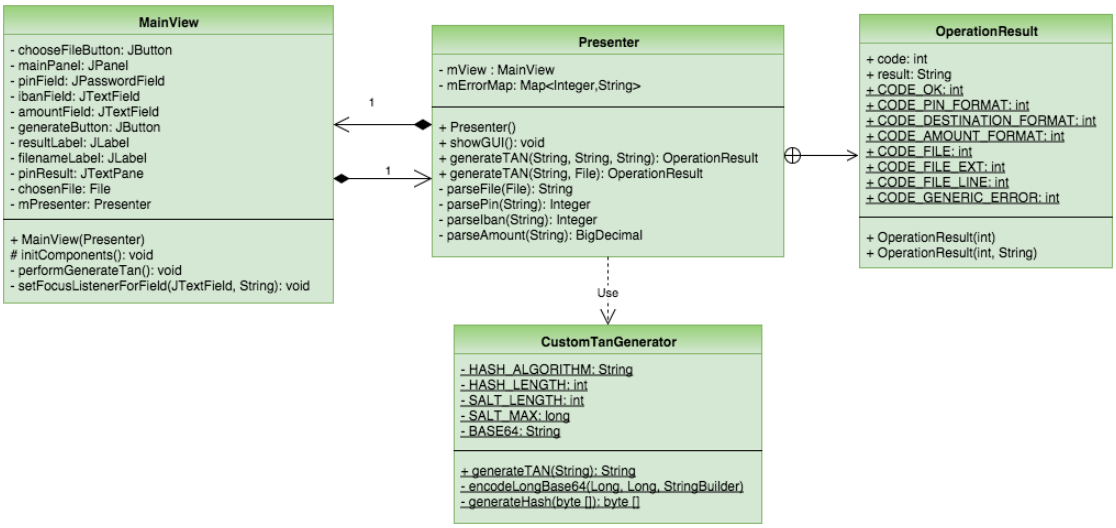


Figure 2.7: Architecture of the SmartCard Simulator

2.3.3 Security considerations

As for any public cryptographic algorithm, the only thing that should be kept private is the secret shared between the client and the server, in this case being the PIN code, therefore the SCS application was not obfuscated in any way. This is also due to the overall low difficulty in reverse-engineering compiled Java code.
The chosen solution is considered to be secure, as the input parameters (namely the batch transaction file or the details of a specific transaction) are hashed with a SHA-256

algorithm, together with the PIN of the user and a timestamp. The purpose of the timestamp is to add a degree of randomness inside the resulting TAN and avoiding replay attacks (the backend stores the time of the last transaction performed by a user). Although the timestamp can easily be guessed by an attacker, all the other values are strictly related to a specific transaction, and a brute-force attack would require a huge amount of attempts to either find a hash collision or the correct values. The Goliath National Bank application also provides a lockout mechanism in case a malicious attacker attempted to brute-force a TAN on the transaction page, locking him out indefinitely.
Since the SCS application does not connect to the Internet in any way and the user is required to manually copy/paste a generated TAN inside the transaction page, an attacker would have to obtain complete access on the victim's machine, in order to obtain a valid TAN code and eventually brute-force the PIN. Since this is highly unlikely and in any case not due to the design of the application, the SmartCard Simulator can be considered secure.

3 Security Measures

Short intro text here

Description: Enumerate the security features your application uses and which attacks each feature defends against (specify which features you implemented and which you borrowed/use from other libraries)

Description	
Implementation	
Secure against	

3.0.1 HTTPS with HSTS

Description	The site is only reachable over HTTPS and employs HSTS (HTTP Strict Transport Security) to protect against a downgrade of future connection attempts to HTTP.
Implementation	The apache2 server was configured in a way that only allows connections via HTTPS and always appends the `Strict-Transport-Security` header to replys to enable HSTS in the clients browser.
Secure against	• Capturing of the communication between the client and the server by an third party. • Modification of the communication between the client and the server by a man-in-the-middle attacker. • Downgrade of a new connection to the server to HTTP by a man-in-the-middle attacker (after an initial connection to the correct site)

3.0.2 SSL secure ciphers

Description	The SSL ciphers offered by the webserver for HTTPS connections are configured to confirm the latest standards.
Implementation	The apache2 server was configured to only offer ciphers that are known secure. As visible in Figure 3.1 and Figure 3.2, current SSL testing tools give the site a perfect score.
Secure against	Save against SSL vulnerabilities. E.g. Heartbleed, POODLE, FREAK. Prevents eavesdropping and man-in-the-middle attacks caused by weak ciphers.

```
--> Testing protocols (via sockets except TLS 1.2 and SPDY/NPN)

 SSLv2      not offered (OK)
 SSLv3      not offered (OK)
 TLS 1      offered
 TLS 1.1    offered
 TLS 1.2    offered (OK)
 SPDY/NPN   not offered

--> Testing ~standard cipher lists

 Null Ciphers                 not offered (OK)
 Anonymous NULL Ciphers       not offered (OK)
 Anonymous DH Ciphers         not offered (OK)
 40 Bit encryption            not offered (OK)
 56 Bit encryption            Local problem: No 56 Bit encryption  configured in /usr/local/opt/openssl/bin/openssl
 Export Ciphers (general)     not offered (OK)
 Low (<=64 Bit)               not offered (OK)
 DES Ciphers                  not offered (OK)
 Medium grade encryption      not offered (OK)
 Triple DES Ciphers           not offered (OK)
 High grade encryption        offered (OK)
```

Figure 3.1: Available SSL Ciphers

```
--> Testing vulnerabilities

 Heartbleed (CVE-2014-0160)                not vulnerable (OK) (timed out)
 CCS (CVE-2014-0224)                       not vulnerable (OK)
 Secure Renegotiation (CVE-2009-3555)      not vulnerable (OK)
 Secure Client-Initiated Renegotiation     not vulnerable (OK)
 CRIME, TLS (CVE-2012-4929)                not vulnerable (OK)
 BREACH (CVE-2013-3587)                    no HTTP compression (OK) (only "/" tested)
 POODLE, SSL (CVE-2014-3566)               not vulnerable (OK)
 TLS_FALLBACK_SCSV (RFC 7507), experim.    Downgrade attack prevention supported (OK)
 FREAK (CVE-2015-0204)                     not vulnerable (OK) (tested with 4/9 ciphers)
 LOGJAM (CVE-2015-4000), experimental      not vulnerable (OK) (tested w/ 2/4 ciphers only!), common primes not checked.
 BEAST (CVE-2011-3389)                     no CBC ciphers for TLS1 (OK)
 RC4 (CVE-2013-2566, CVE-2015-2808)        no RC4 ciphers detected (OK)
```

Figure 3.2: No known SSL Vulnerabilities

3.0.3 CSRF tokens

Description	The application automatically creates anti-CSRF cryptographic nonces (tokens) on all pages with forms that could be exploited to perform operations using the profile of a user. These tokens are validated once the form is submitted by the user, ensuring that the application can't be forced to perform actions via cross-site requests.
Implementation	This feature was implemented manually, by creating a 256-bit random token (obtained via the PHP builtin `openssl_random_pseudo_bytes` function) and encoding it base64. This token is then saved as a session variable and sent to the user as a hidden field inside a form, as can be seen in the figure below. Once the form is submitted by the user, the the token contained inside the form is compared against the one already existing on the server.
Secure against	This particular security feature protects against XSRF attacks, as described in OTG-SESS-005.

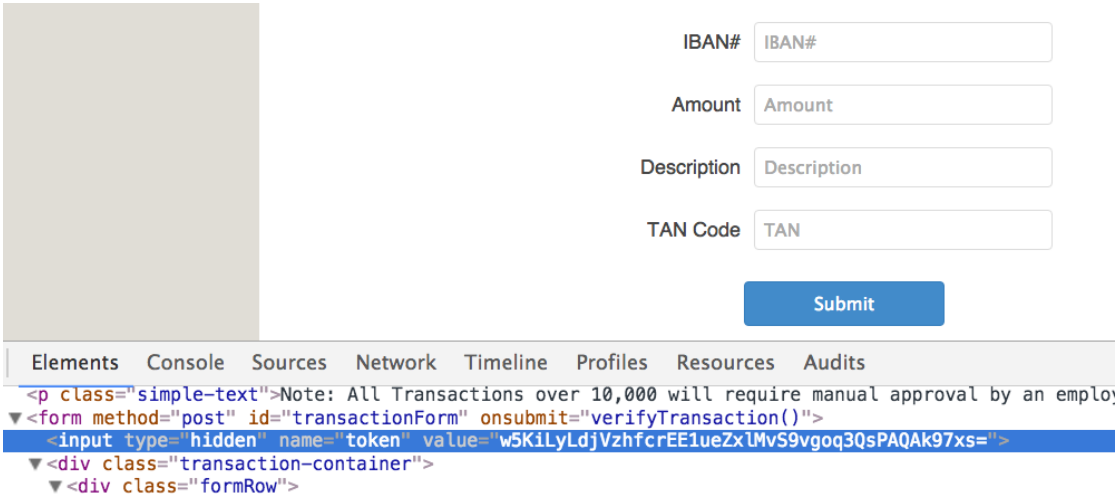


Figure 3.3: CSRF token generated in the transaction page

3.0.4 Password strength

Description	We enforce strong passwords by forcing the users to respect the following criteria: • length between 8 and 20 characters; • at least 1 numeric character (0-9); • at least 1 upper/lower case letter (a-zA-Z); • any special character inside the password is allowed. Any password not matching these criteria is rejected by the server.
Implementation	The password enforcing mechanism was manually implemented both on client side (see `project/js/registration.js`) and on server side (see `project/registration/registration_request.php`). The user is compelled to follow the mentioned criteria when choosing a password during registration. This also holds true when resetting a password later on.
Secure against	Strong password prevent attackers from guessing them, as described in OTG-AUTHN-007.

3.0.5 Secure passwords

Description	User passwords are stored inside the database after a successful registration. Passwords are never stored in plaintext, but are hashed using a random salt, unique for every user, together with an additional static salt, known only to the backend server.
Implementation	The algorithm used for hashing these values together is the SHA-512. A salt is generated for every user via the builtin PHP `openssl_random_pseudo_bytes` function. The salt and the hashed password are both stored on the database (see section 2.2). No additional libraries were used to implement this feature.
Secure against	Secure passwords protect the users in case the database is compromised in any way. An attacker is not able to determine which users are using the same password and cannot retrieve the original password, starting from the hashed one.

3.0.6 Password reset with two-factor authentication

Description	The user has the possibility to reset his password in case he does not remember it. A password reset requires the users mail address and access to the used mail account. Additionally the user has to enter his PIN to complete the process of resetting his password. This prevents attackers that already have access to the users mail address from resetting the password.
Implementation	This has been implemented using the columns `pw_reset_hash`, `pw_reset_hash_timestamp` and `pin` in the table `user`. When a password reset is requested a mail is sent to the user with a randomly generated token which is also stored in the database with the current timestamp. The user then has to use this token before it is invalidated by clicking on the link in the email, and providing a new password and the current PIN in the form.
Secure against	This security feature protects users from attackers that already have access to their mail account, physical access to their device(s) or started the password reset process through social engineering.

3.0.7 Lockout mechanism after failed password entries

Description	In order to use our website users are required to log in. For a successful login the right username (which is the users mail address) and the correct password are required. To prevent brute force attacks given an already known mail address a lockout mechanism has been implemented. This mechanism blocks the user account if the wrong password has been entered 5 times in a row. The user has to be unblocked by an employee afterwards. The same functionality was implemented to prevent brute force attacks on TANs for transactions.
Implementation	This feature was implemented by adding the table `failed_attempts` to the database (see section 2.2). This table stores all failed attempts (e.g. login or an invalid TAN) with the timestamp of the attempt and the ID of the user that executed the action. Every time an invalid password or TAN is observed a failed attempt is added the database and the sum of all failed attempts is checked. Once a valid password or TAN is entered all failed attempts are reset.
Secure against	This feature protects against Application Mis-Use (OTG-BUSLOGIC-007) and brute force attacks on the login functionality (OTG-AUTHN-003).

3.0.8 Automatic logout after inactivity

Description	The application automatically logs out users who have been inactive for 30 minutes.
Implementation	This was implemented on two different levels. The `session.gc_maxlifetime` setting in the php configuration allows sessions to be removed after 1440 seconds. Additionally the user is logged out by a check in the php code if it latest activity lies more than 30 minutes in the past.
Secure against	Prevents stealing of the user session from the clients computer after the timeout.

3.0.9 CAPTCHAs

Description	During the registration process users are required to enter a CAPTCHA code in order to prevent malicious attackers from automating registrations. This CAPTCHA is entirely random and requires the user to read an image and input the alphanumeric code manually.
Implementation	For the creation of the CAPTCHA we resort to the `secureimage` library (see 2 for further info).
Secure against	This security feature protects the application from attackers who try to automate registrations (see OTG-IDENT-002), which could lead to a database saturation or even to a DOS.

3.0.10 Balance calculated in database (Realtime concurrent transactions)

Description	As described in section 2.2 the balance for each given account is always calculated in realtime in the database. The `VIEW account_overview` uses all existing transactions and determines its value for the given account using the transaction status, the destination account and source account. This not only ensures an always consistent database because it is not possible to remove the balance for one account and "forget" to update the balance for the other account, but also enables realtime concurrent transactions, because no locking is necessary when new transactions are added.
Implementation	This feature was implemented using the SQL-`VIEW account_overview` which combines the tables `account` and `transaction` to calculate the current balance for a given account. The implementation of this `VIEW` is shown in Figure 2.5 and explained in section 2.2.
Secure against	This security feature protects against flaws in the business logic data validation (OTG-BUSLOGIC-001) e.g. if actions are not completed successfully without taking necessary rollback steps. This also prevents exploiting a possible Circumvention of Work Flow (OTG-BUSLOGIC-006) as one transaction is an atomar operation. Summed up this feature ensures that the database is always consistent and no money can get lost due to incompletely handled transactions.

3.0.11 Password protected PDFs for TAN lists

Description	The TAN list sent out to new users upon approval is encrypted with a password which gets sent to the user over a separate channel.
Implementation	We encrypt the TAN list PDF with a password derived from the user pin. The user gets this password displayed in the webinterface when he successfully logs in.
Secure against	Interception of the TAN list as it's sent over an unencrypted channel (email).

3.0.12 Time-based TANs generated via SCS

3.0.13 Secure cookies

Description	
Implementation	
Secure against	

3.0.14 Input sanitization

prevents XSS, command injection and sql injection

Description	
Implementation	
Secure against	

3.0.15 Prepared statements

Description	
Implementation	
Secure against	

3.0.16 Clickjacking prevention

X-Frame-Options

Description	
Implementation	
Secure against	

4 Fixes

4.1 Configuration and Deploy Management Testing

4.1.1 Test File Extensions Handling for Sensitive Information (OTG-CONFIG-001)

Vulnerability	When performing batch transactions, the transaction file uploaded by the user was accessible by everyone who could guess the filename until the transaction was processed.
Original CVSS	**AV:** N **AC:** H **PR:** N **UI:** R **S:** U **C:** L **I:** N **A:** N **Score:** 3.1
Countermeasures	To prevent attackers from accessing sensitive information we now prevent all access to the lib and tmp folders by denying access to them in the apache2 site configuration. PHP files that shouldn't be accessed directly by the user are now located in the lib folder. The newly created tmp folder now contains the previously mentioned uploads folder for temporary batch transaction files and the holder folder for temporary password protected tan lists of new users. Additionally the same error page is shown for "not found (404)" and "permission denied (403)" errors to prevent disclosing the internal folder structure.

Modified file(s)	`/config/apache2/site-available/gnb`
	`/project/resource_mappings.php`
	`Moved files/directories:`
	• `/project/uploads/ -> /project/tmp/uploads/`
	• `/project/holder/ -> /project/tmp/holder/`
	• `/project/gnbmailer.php -> /project/lib/gnbmailer/gnbmailer.php`
	• `/project/templates/ -> /project/lib/gnbmailer/templates/`

Modified line(s)	Added/changed lines 21-34 in the `gnb` apache2 config file Updated lines 33, 34, 56, 63, 64 in the `resource_mappings.php` to reflect the changed paths

4.2 Identity Management Testing

4.2.1 Test User Registration Process (OTG-IDENT-002)

Vulnerability	The registration process was available to anyone, allowing to register countless users as long as a valid email was provided. Although a user would need to be manually approved by an employee of the bank, it was possible to generate DOS attacks by creating robot accounts (or at least saturating the database with dummy data).
Original CVSS	**AV:** N **AC:** L **PR:** N **UI:** N **S:** U **C:** N **I:** N **A:** L **Score:** 5.3
Countermeasures	To prevent attackers to register countless users using automated scripts we added a CAPTCHA functionality.
Modified file(s)	`/project/registration/registration.php`, `/project/registration/registration_request.php`
Modified line(s)	Added lines 13, 96-106 inside `registration.php`, and added lines 25, 35, 56-61 inside `registration_request.php`.

4.2.2 Test Account Provisioning Process (OTG-IDENT-003)

Vulnerability	Provisioning clients is an easy process with no effective mechanisms to verify or vet clients besides a manual approval process, provisioning employees is set up in a similar matter.
Original CVSS	**AV:** N **AC:** L **PR:** N **UI:** N **S:** U **C:** N **I:** N **A:** L **Score:** 5.3
Countermeasures	Only employees/admins are allowed to approve or reject user registrations. Since this application is only web-based and cannot provide any out of bound verification (e.g. the personal ID of a client, his tax code or similar), we must presume this can only happen physically at the Bank. Also the assumption is that users (both clients and employees) will only be approved by employees after a meeting in person at the bank, during which an employee has verified the personal data of said user. Given this argument, we decided to treat the account provisioning process as secure.
Modified file(s)	`None`
Modified line(s)	N/A

4.3 Error Handling

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

Vulnerability	When uploading a batch file containing format errors the application would return an error code instead of an error message.
Original CVSS	**AV:** P **AC:** L **PR:** L **UI:** R **S:** U **C:** L **I:** N **A:** N **Score:** 1.9
Countermeasures	The vulnerability was due to the application displaying all messages returned from the C parser to the client. We simply changed the output of the parser to display a custom error, containing the line of the batch file in which the error occurred, instead of an error code.
Modified file(s)	`/project/lib/ctransact/src/ctransact.c`
Modified line(s)	Modified line 65 inside `ctransact.c`

4.4 Cryptography

4.4.1 Testing for Weak SSL/TSL Ciphers, Insufficient Transport Layer Protection (OTG-CRYPST-001)

Vulnerability	The web server configuration offered SSL ciphers which were vulnerable against POODLE and RC4.
Original CVSS	**AV:** N **AC:** H **PR:** N **UI:** R **S:** U **C:** L **I:** N **A:** N **Score:** 3.1
Countermeasures	We updated the apache2 webserver configuration to disable all insecure ciphers. The current configuration is documented subsection 3.0.2
Modified file(s)	`/config/apache2/httpd.conf`
Modified line(s)	Changed lines 1-7 of `httpd.conf`

4.5 Business Logic Testing

4.5.1 Test Business Logic Data Validation (OTG-BUSLOG-001)

Vulnerability	The application allowed transaction to the account `10000001` which belongs to Barney Stinson, who is the admin of our bank system. As defined in our database (see section 2.2) Barney always has an account balance of `1.000.000.000`. Due to this fact transactions to barney decrease the balance of the source account while not affecting the destination account. This also leads to mysterious "loss" of money because the bank start page shows the total amount of money on all our accounts - excluding barneys account.
Original CVSS	**AV:** N **AC:** L **PR:** L **UI:** R **S:** U **C:** N **I:** H **A:** N **Score:** 5.7
Countermeasures	To avoid this issue it is no longer possible to execute transaction to account number `10000001`. This has been implemented using simple checks for the account number in both the single transactions process as well as the batch transactions parser.
Modified file(s)	`/project/accounts/verify_transaction.php`, `/project/lib/ctransact/src/ctransact.c`
Modified line(s)	Added/changed lines 147, 1213-1216, 1319-1323 and 1344 in `verify_transaction.php` and lines 113-117 in `ctransact.c`

4.5.2 Testing for the Circumvention of Work Flows (OTG-BUSLOG-006)

Vulnerability	The application was generating plain error messages upon a login attempt with an empty password field, preventing the user from returning to the login page via graphical means, and providing some insight about the application's error handling policy.
Original CVSS	**AV:** N **AC:** L **PR:** L **UI:** R **S:** U **C:** N **I:** N **A:** L **Score:** 3.5
Countermeasures	The Goliath National Bank application returns custom error messages when submitting forms with user input. As of phase 3, user input was sanitized using a custom `sanitize_input` function. This function would simply exit, returning a generic error message (without html formatting), in case an input wasn't sanitized correctly. This was due to a faulty error message handling, which was later on discovered on other pages as well. We modified the input sanitization function (the `check_post_input` function was also added) to return a value in case an input wasn't sanitized correctly (instead of exiting). This allowed to generate more specific errors, depending on the page. Furthermore, by slightly changing the input checks inside other pages, we implemented a simpler logic for displaying error messages. For uniformity in error reporting, we applied minor changes to the following files as well: `new_transaction.php`, `verify_transaction.php`, `new_transaction_multiple.php`, `registration.php`, `registration_request.php`.
Modified file(s)	`/project/genericfunctions.php`, `/project/authentication.php`, `/project/login.php`
Modified line(s)	Added lines 10-15 inside `genericfunctions.php` and accordingly edited lines 15-21 inside `authentication.php`, as well as lines 60-69 inside `login.php`.

4.5.3 Test Defenses Against Application Mis-use (OTG-BUSLOG-007)

Vulnerability	The application did not implement any mechanisms to prevent against application mis-use except the Lockout-Mechanism in face of too many failed login attempts. This enabled attackers to brute-force TANs for single as well as multiple transactions.
Original CVSS	**AV:** N **AC:** L **PR:** N **UI:** N **S:** U **C:** L **I:** L **A:** L **Score:** 7.3
Countermeasures	The application now keeps track of failed login attempts as well as invalid transaction numbers (TANs). This is realized using a table that can flexibly count all kinds of failed attempts associated with an user ID. That means that this mechanism can easily be extended to keep track of additional mis-uses of the application. Too many invalid TANs now lead to a logout of the user and blocking of the account. The account then has to be unblocked by an employee.
Modified file(s)	`/project/accounts/verify_transaction.php`, `/project/db.php`, `/project/logout.php`, `project/accounts/new_transaction_multiple.php`, `config/database/gnbdb_create.sql`
Modified line(s)	Added/changed lines 57, 107 and 117 in `verify_transaction.php` Added/changed lines 80, 176-187, 284, 316-430, 744-746, 945-946, 1204, 1250, 1295, 1303 and 1344 in `db.php` Added line 10 in `logout.php` Added/changed lines 102, 105 and 110-117 in `new_transaction_multiple.php` Added/changed lines 2, 143, 145, 148, 150 and 152 in `gnbdb_create.sql`