Secure Coding WS14/15

Team 20 Phase 4

Presented by

Michael Remmler
Bharti Munjal
Simon Barth

Executive Summary

In this report we talk about the vulnerabilities we have found in banana bank, team 2 and in our own application. The report discusses each of the OWASP Testing Guide items for both apps - our observations, discovery steps along with any likelihood and impact.

We found several vulnerabilities of high risk in team 2 VM. These vulnerabilities range from making transactions using any random tan value to Cross Site Request Forgery, Command Injection, Directory Listing and ClickJacking. Moreover we found that the application sends sensitive data over unencrypted network by using HTTP instead of HTTPS. However the web part of team 2 application is successfully protected against XSS attacks and SQL Injections.

The white box analysis of team 2's code also revealed that bank uses one component of the phpsec framework to encrypt the database information. Some other components of the framework are also present but not used. The structure of the source code doesn't follow object oriented design principles. We could find several examples for redundant code that is copy and pasted to several locations.

Our own VM showed no vulnerabilities against SQL nor against XSS/CSRF in our tests. Overall the vulnerabilities we found in our application are minor problems like cookie attributes, the missing HSTS header flag and unexpected file upload. The major problems are the weak password change policy, browser cache policy and the default credentials for the database combined with Adminer accessibility.

Table of Content

Time Tracking Table

Task	Student	Workload
Project Management	Each	4h
Creating Presentation + Videos	Each	3h
Working through Checklist, php analysis, writing report	Michael	15h + 20h + 30h = 65h
Working through Checklist, Java reverse engineering, writing report	Bharti	25h + 7h + 20h = 52h
Working through Checklist, C reverse engineering, writing report	Simon	15h + 10h + 35h = 60h

Overview

For the evaluation of the severity of vulnerabilities we used the following matrix:

Likelihood	Impact	Risk
LOW	LOW	LOW
LOW	MEDIUM	LOW
LOW	HIGH	MEDIUM
MEDIUM	LOW	LOW
MEDIUM	MEDIUM	MEDIUM
MEDIUM	HIGH	HIGH
HIGH	LOW	MEDIUM
HIGH	MEDIUM	HIGH
HIGH	HIGH	HIGH

The most severe vulnerabilities we could identify in the banana bank application are listed in the following table:

Name	Phase	Likelihood	Impact	Risk
Command Injection	4	LOW	HIGH	MEDIUM
Directory Listing	2 & 4	HIGH	HIGH	HIGH
Strict HTTPS	2 & 4	HIGH	HIGH	HIGH
Clickjacking	4	MEDIUM	HIGH	HIGH
CSRF	4	MEDIUM	HIGH	HIGH
Use any TAN in SCS mode	4	HIGH	HIGH	HIGH
Download history and scs of any client	2 & 4	HIGH	MEDIUM	HIGH
Downloading the C program	4	HIGH	MEDIUM	HIGH
Number of found vulnerabilities				24

The most severe vulnerabilities we could identify in our application are listed in the following table:

Name	Phase	Likelihood	Impact	Risk
Weak Change password policy	4	MEDIUM	HIGH	HIGH
Default Credentials	4	MEDIUM	HIGH	HIGH
Weak browser cache policy	4	MEDIUM	HIGH	HIGH
Cookie Attributes	2 & 4	LOW	HIGH	MEDIUM
Upload of Unexpected File Types	2 & 4	MEDIUM	HIGH	HIGH
Number of found vulnerabilities				13

Unfixed Vulnerabilities of Phase 2 of banana bank

HTTP/HTTPS

Even though HTTPS is now an option, the application is not forcing it so users can still access the web site via unencrypted channels (see “Authentication Testing (4.5)” - “Testing for Credentials Transported over an Encrypted Channel (OTG-AUTHN-001)”)

Session Management

As reported in phase 2, the user is redirected to login page on click of "Home Page" link. This gives an impression to the user that he has been logged out but in reality the session is live and thus posing threat on security.

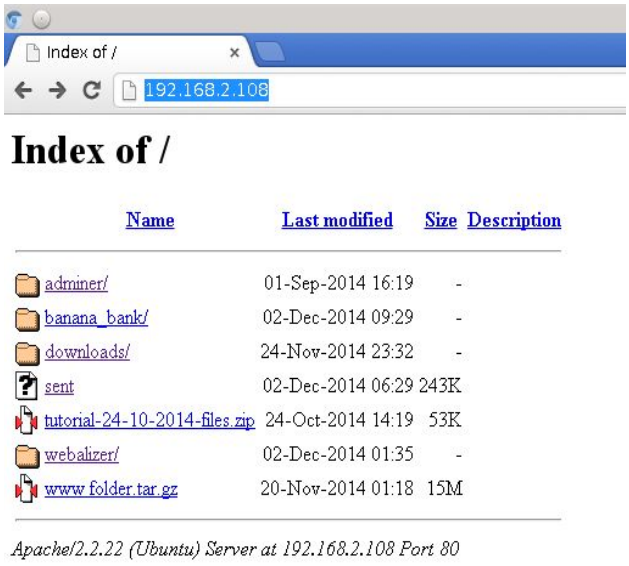
Directory Listing

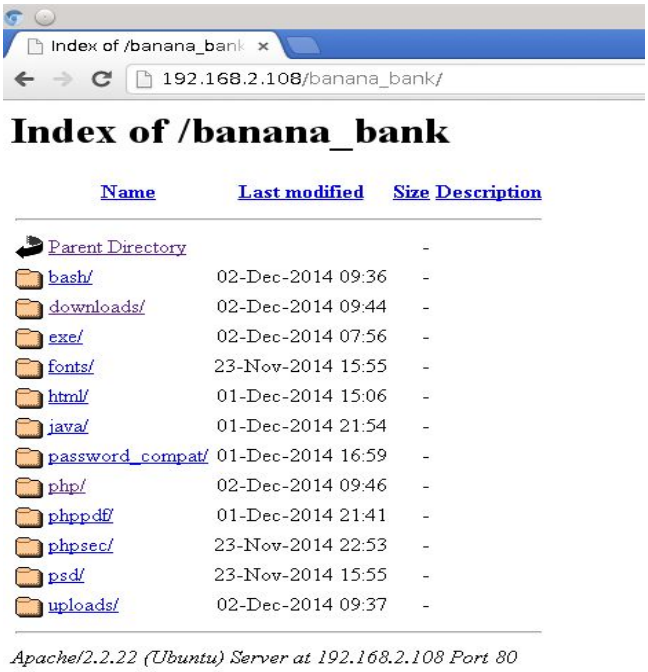
Likelihood:	HIGH	Impact:	HIGH	Risk:	HIGH

Observation:

It is possible to have a look at parts of the file structure of the websites. So its for example possible to download the transaction batch files.

Discovery:





Likelihood:

There are no technical skills required other than using a web browser.

Implication:

If any of the files that are exposed contains sensitive information like credentials, db-layout or similar, an attacker could gain critical information about the server and application from exploiting this vulnerability.

Recommendation:

Use access restriction on directories and file types and remove all unnecessary files and folders.

Details of OWASP Checklist

Configuration and Deploy Management Testing (4.3)

Test File Extensions Handling for Sensitive Information (OTG-CONFIG-003)

Other Application:

Likelihood:	HIGH	Impact:	HIGH	Risk:	HIGH

Observation:

All PHP files on the server were properly processed by the webserver before serving them. Therefore no source code could be read outside the server. However other file extensions like XML, sh, PSD, MD and TXT are served without filtering or preprocessing by the server.

Discovery:

1. Go to http://<IP-ADDRESS>/banana_bank/php
2. Try to download any file
3. View page source
    a. It's an empty file

1. Go to http://<IP-ADDRESS>/banana_bank/bash
2. Try to download cleaner.sh
3. The file can be viewed as a normal text file

The discovery for file types MD, PSD and XML is similar, only the directories are different. The other directories are 'psd', 'java', 'phpsec/auth' and 'phppdf'.

Likelihood:

There are no technical skills required other than using a web browser. Therefore the probability of exploiting a vulnerability in file extension handling is high.

Implication:

If any of the files that are exposed contains sensitive information like credentials, db-layout or similar, an attacker could gain critical information about the server and application from exploiting this vulnerability.

Recommendation:

Restrict access to certain directories or move sensitive files to locations that are not accessible from the web if they don't need to be accessed.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

There are no files directly accessible. The URL gets rewritten and file names are not exposed. There were no unfiltered file extensions.

Backup and Unreferenced Files for Sensitive Information (OTG-CONFIG-004)

Other Application:

Likelihood:	LOW	Impact:	LOW	Risk:	LOW

Observation:

Inside the `html` directory of the `banana_bank` web application there are two temporary files that probably have been left by an editor. The files are `clientInitial.html~` and `functions.js~`. We also found a shell script in a directory called `bash`.

Discovery:

We tried normal file names we knew from using the website with backup extensions common to some editors. The shell script was revealed using the spider in ZED proxy.

Likelihood:

Since the server is also vulnerable against directory listings, all this can be easily found.

Impact:

All files that we found did not contain any confidential information. There were no php backup files which might have given away the source code since they wouldn't be processed by the php interpreter of apache.

Recommendation:

Be sure to not leave any backup files around. Also check to restrict access to directories.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

ZED couldn't find any files and we also didn't find any backup files.

Enumerate Infrastructure and Application Admin Interfaces (OTG-CONFIG-005)

Other Application:

Likelihood:	HIGH	Impact:	HIGH	Risk:	HIGH

Observation:

We could access the Adminer database administration webinterface.

Discovery:

It's the standard URL http://<IP>/adminer

Likelihood:

Anybody with a web browser can find the database administration page.

Impact:

If an attacker gets access to can guess the password, all databases, not only the one for banana_bank can be compromised.

Recommendation

Remove any administration web interfaces from the document root, for example by changing the server's document root to your applications html directory.

Our Application:

Likelihood:	HIGH	Impact:	HIGH	Risk:	HIGH

Observation:

The VM we handed in in phase 3 had the adminer web interface available as well.

Test HTTP Methods (OTG-CONFIG-006)

Other Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

The web server used by banana bank only allows the HTTP methods GET, HEAD, POST and OPTIONS.

Discovery:

Using nikto:

```
nikto -Plugins httpoptions -h <IP-ADDRESS>
```

outputs:

```
samurai@samurai-wtf:program$ nikto -Plugins httpoptions -h 192.168.1.220
- Nikto v2.1.4
---------------------------------------------------------------------------
+ Target IP:          192.168.1.220
+ Target Hostname:    samurai-wtf.lan
+ Target Port:        80
+ Start Time:         2015-01-10 00:43:44
---------------------------------------------------------------------------
+ Server: Apache/2.2.22 (Ubuntu)
+ Allowed HTTP Methods: GET, HEAD, POST, OPTIONS
+ 6456 items checked: 0 error(s) and 1 item(s) reported on remote host
+ End Time:           2015-01-10 00:43:45 (1 seconds)
---------------------------------------------------------------------------
+ 1 host(s) tested
```

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

The web server used by banana bank only allows the HTTP methods GET, HEAD, POST and OPTIONS.

Discovery:

Using nikto:

`nikto -Plugins httpoptions -h <IP-ADDRESS>`

outputs:

```
samurai@samurai-wtf:program$ nikto -Plugins httpoptions -h 192.168.1.131
- Nikto v2.1.4
---------------------------------------------------------------------------
+ Target IP:          192.168.1.131
+ Target Hostname:    samurai-wtf.local
+ Target Port:        80
+ Start Time:         2015-01-10 00:44:22
---------------------------------------------------------------------------
+ Server: Apache/2.2.22 (Ubuntu)
+ Allowed HTTP Methods: GET, HEAD, POST, OPTIONS
+ 6456 items checked: 0 error(s) and 1 item(s) reported on remote host
+ End Time:           2015-01-10 00:44:23 (1 seconds)
---------------------------------------------------------------------------
+ 1 host(s) tested
```

Test HTTP Strict Transport Security (OTG-CONFIG-007)

Other Application:

Likelihood:	MEDIUM	Impact:	MEDIUM	Risk:	MEDIUM

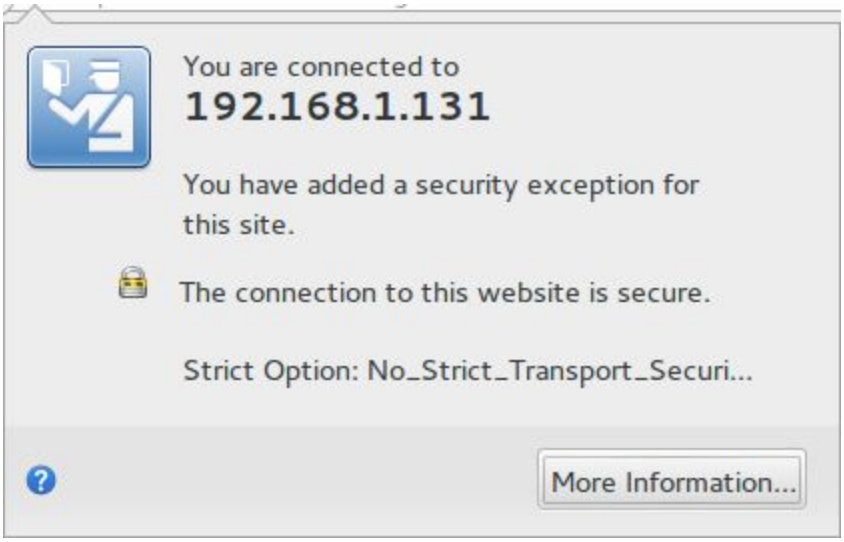
Observation:

The banana bank web application is not using HTTP strict transport security.

Discovery:

1. Firefox Add-on “Strict Transport Security - Declarative Security Detector”
2. Visit banana bank web application via https

3. Check output of add-on:



Likelihood:

An attacker needs medium to high knowledge of HTTP and HTTPS as well as sniffing and other attacks in order to exploit this vulnerability

Impact:

Exploitation of this vulnerability can be used for:

- Information sniffing of unencrypted traffic
- Man in the middle attacks

Recommendation:

Activate strict option in https headers. This can be done using the 'mod_headers' module in Apache and configure it to add the strict transport security field in the header of https requests as described here: http://blog.nvisium.com/2014/04/is-your-site-hsts-enabled.html.

Our Application:

Likelihood:	MEDIUM	Impact:	MEDIUM	Risk:	MEDIUM

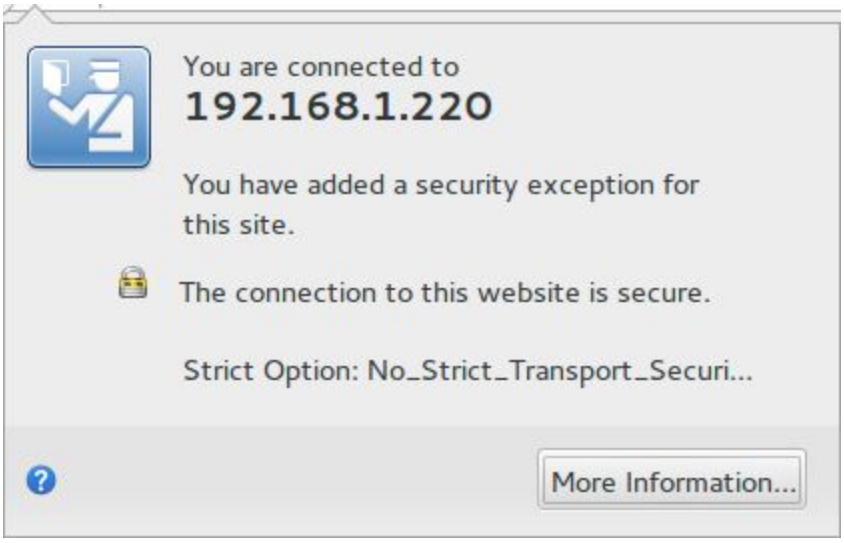
Observation:

The banana bank web application is not using HTTP strict transport security.

Discovery:

1. Firefox Add-on "Strict Transport Security - Declarative Security Detector"
2. Visit banana bank web application via https

3. Check output of add-on:



Likelihood, Impact and Recommendation are exactly the same as for banana bank.

Identity Management Testing (4.4)

Test Role Definitions (OTG-IDENT-001)

Expectation:

The following role matrix shows the expected activities a role can do.

Role	Register Client	Register Employee	Accept Client Account	Accept Employee Account	Set Client Balance	Create Transaction	View own balance	View others balance	Forgot Password
Logged in Employee	NO	NO	YES	YES	YES	NO	NO	YES	YES
Logged in Client	NO	NO	NO	NO	NO	YES	YES	NO	YES
Logged out Client or Employee	YES	YES	NO	NO	NO	NO	NO	NO	YES
Unregistered User	YES	YES	NO	NO	NO	NO	NO	NO	NO

Other Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

The roles as implemented in the banana bank application were different from the expected actions in the following points:

- logged in employees and clients can register
- logged in employees and clients can log in

Discovery:

To check for allowed and restricted actions:

1. Use the page in each role, try to execute actions that are expected to be possible and note the pages included in these actions.
2. Try to access pages from a role that are part of an action for another role, e.g. access clientNewTransaction.html as an employee.
3. Check if this is possible, if there are errors and which functionality is usable on that page.

To check for different behaviour from matrix:

1. Log in as employee or client
2. Click on ‘Home Page’
    a. Login as client or

b. Click on ‘Employees’ and log in as an employee or
c. Click on ‘Password forgotten’ and request a new password

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

The roles were restricted to their actions and allowed all expected activities, no vulnerability was found.

Discovery:

1. Login in any role
2. Execute actions expected to work and note URL belonging to action
3. Login as different role and try to access URLs that belong to actions that should not be accessible
4. Try to access URLs that belong to actions that should be inaccessible without being logged in
5. Try to access URLs for actions that should be only accessible without being logged in (register, log in) while logged in as employee or client.

Test User Registration Process (OTG-IDENT-002)

Other Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

Users can register and receive an email with their encrypted TAN list or SCS information. A new account cannot login before being provisioned. It is not possible to register an employee with the same email address as a client.

Discovery:

1. Create a new user account
2. Try to login with new account

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

Users can register and receive an email with their encrypted TAN list or SCS information. A new account cannot login before being provisioned. It is not possible to register an employee with the same email address as a client.

Discovery:

1. Create a new user account
2. Try to login with new account

Test Account Provisioning Process (OTG-IDENT-003)

Other Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

New user accounts cannot login unless the account was provisioned by an employee.

Discovery:

1. Register new user account
2. Try to login -> Not possible
3. Login as employee
4. Accept account
5. Logout employee
6. Login with new user account created in step 1 -> Account can login

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

New user accounts cannot login unless the account was provisioned by an employee.

Discovery:

1. Register new user account
2. Try to login -> Not possible
3. Login as employee
4. Accept account
5. Logout employee
6. Login with new user account created in step 1 -> Account can login

Testing for Account Enumeration and Guessable User Account (OTG-IDENT-004) and Testing for Weak or unenforced username policy (OTG-IDENT-005)

Other Application:

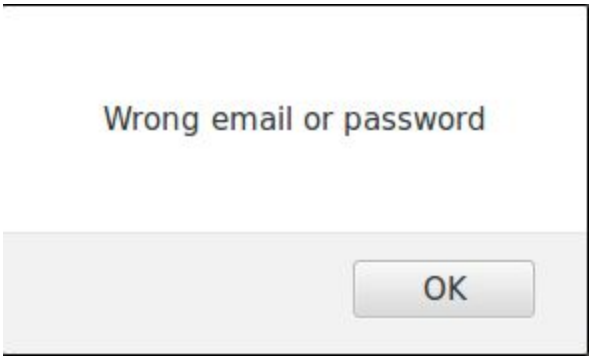
Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

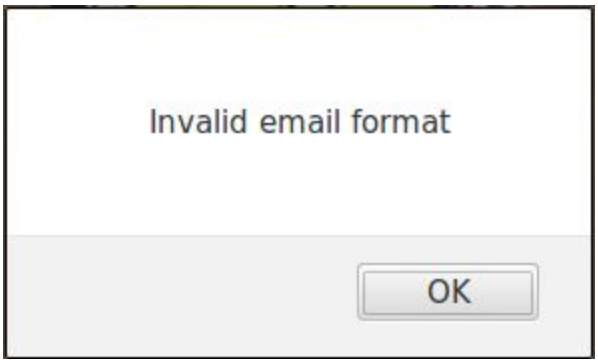
You cannot tell from the error message at login if an account exists and you just mistyped the password or if the account doesn't exist at all. Account names are the email address and therefore guessable if an attacker knows the email address of the victim.

Discovery:

1. Try logging in an existing account 'client1@mybank.de' with wrong password 'p'
2. Following error message is shown:



3. Try logging with an account that doesn't exist 'abcde' and any password 'p'
4. Following error message is shown:



Our Application:

Likelihood:	HIGH	Impact:	LOW	Risk:	LOW

Observation:

It is possible to tell if an account exists by the error message of an unsuccessful login. The application uses usernames that a user can choose for login and forgotten password functionality. Knowing the email address of a victim doesn't help an attacker for breaking the login protection.

Discovery:

1. Try to log in with an existing account and wrong password, e.g. 'user1' pw: 'p'
2. The following error message will be displayed:

   Incorrect Username/Password combination!

3. Try to log in with an account that doesn't exist and any password
4. The following error message will be displayed:

   This User does not exists!

Likelihood:

An attacker needs only little technical knowledge to use a web browser to guess account names.

Impact:

Knowing the account name an attacker only needs to brute force the password. Since the application enforces strong passwords and has brute force protection by limiting the number of login attempts per time unit, the impact is low.

Recommendation:

Use the same error message for non existing accounts and wrong credentials for an existing account.

Authentication Testing (4.5)

Testing for Credentials Transported over an Encrypted Channel (OTG-AUTHN-001)

Other Application:

Likelihood:	HIGH	Impact:	HIGH	Risk:	HIGH

Observation:

We observed that it is possible to access the webpage via HTTP and HTTPS.

Discovery:

1. browse to https://<IP-Address>/banana_bank/html/
2. login into the bank as a customer (acc: client1@mybank.de, pw:Please1)
3. you will be redirected to https://<IP-Address>/banana_bank/html/clientInitial.html
4. browse to http://<IP-Address>/banana_bank/html/clientInitial.html

Likelihood:

Exploitation of this vulnerability requires no advanced technical skills.You just need to change the url from https to http.

Impact:

If the customer uses http-urls all his data is send via unencrypted change. This could lead to sensitive data exposure. This is very critical if the user's authentication data is send by the login form.

Recommendation:

Disable HTTP for your Application.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

We enabled HTTPS for our whole application. If you are trying to access the pages with HTTP you will be redirected to HTTPS or get an page not found error. For example http://<IP-Address>/bank is redirected to https://<IP-Address>/login. If you are trying to access http://<IP-Address>/customers/createTransaction you will get page not found.

Testing for default credentials (OTG-AUTHN-002)

Other Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

We observed that not only the banana bank application (https://<IP-Address>/banana_bank/html/) is accessible but also the sql webfrontend adminer (http://<IP-Address>/adminer/). We tested default credentials like:

- account: admin ; password: admin
- account: admin ; password: samurai
- account: root ; password: root
- account: root ; password: samurai

But we didn't get access to the applications.

Our Application:

Likelihood:	MEDIUM	Impact:	HIGH	Risk:	HIGH

Observation:

We observed that the bank application (https://<IP-Address>/bank) and adminer (http://<IP-Address>/adminer) is accessible. We tried the same default credentials as for the other vm. We were able to successfully log into adminer with the following credentials account:root with password: samurai.

Likelihood:

Exploitation of this vulnerability requires no advanced technical skills. The username root is a very common default username. The password samurai isn't a common default password, but if the attacker uses the dictionary for a brute force attack, he could find it out.

Impact:

An attacker could manipulate the whole application.

Recommendation:

We need to change the password of the root user in the database and ideally deactivate the adminer web interface.

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

Other Application:

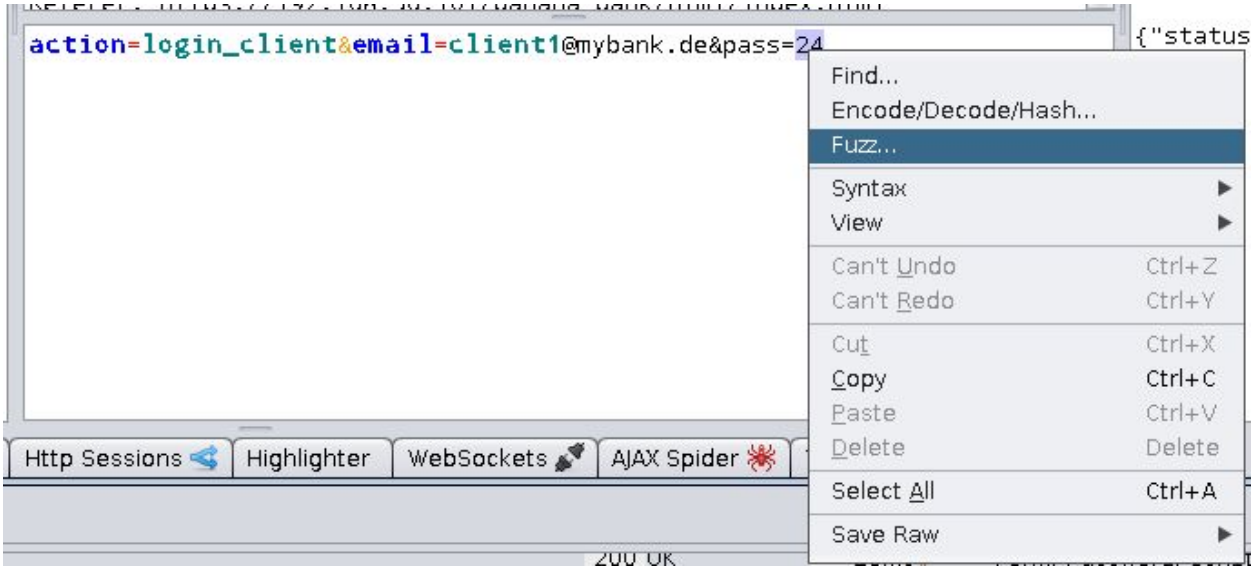
Likelihood:	HIGH	Impact:	HIGH	Risk:	HIGH

Observation:

By checking the code of the login function (login_client-function in client_api.php line 45ff or login_emp-function in employee_api.php line 37-67) , we couldn't find any protection against password brute force attacks. They only did some format checking on the email address. After that they call the login_client_db/login_emp_db-function. These functions only check if the user exists and if the given password is correct.

Discovery:

1. use ZAP to intercept all requests
2. try to login as an client or employee
3. open ZAP fuzzer to brute force the password



4. select a word list, click fuzz and wait until the fuzzer has finished

5. now login with the correct password
6. its possible to login. the account isn't locked

Likelihood:

It is very easy and obvious to test for this vulnerability. You don't even need to use any tools or technical skills.

Impact:

If an attacker manages to find out your password. He could log into your account, access sensitive account information and perform actions like transferring money.

Recommendation:

Use the phpsec framework or implement a mechanism which logs wrong password attempts and locks the account if the attempt count is greater than for example 5 or 10.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

We are using the phpsec framework which has the functionality of detecting brute force attacks. It will detect an attack if one of the two conditions are true:
- 2 failed logins within 1 second
- 5 failed logins within 25 seconds

Than the account is locked and the user needs to perform a forgot password action.

The login functionality is handled by the UsersLoginController (phpsec / framework / control / users / UsersLoginController.php, lines 19-41). It tries to create an object of the phpsec class AdvancedPasswordManagement. Inside the constructor it calls the isBruteForce class function to detect if there is an brute force attack or not (phpsec / libs / auth / adv_password.php, lines 128-168).

Testing for bypassing authentication schema (OTG-AUTHN-004)

Other Application:

Likelihood:	HIGH	Impact:	MEDIUM	Risk:	HIGH

Observation:

Each action, which could be requested by a user, checks if the access is granted or not. If access is proved or not depends on two conditions:

1. a valid and active session exists (client/employee has been logged in)
2. the session variable 'is_employee' is set (to true or false respectively) when an employee or client logs in.

All action for clients except of login_client and reg_client are protected with:

```
if ($_SESSION['is_employee'] == 'true')
    return error('Invalid operation for employee');
```

All action for employees except of login_emp and reg_emp are protected with:

```
if ($_SESSION['is_employee'] == 'false')
    return error('Unauthorized operation for client');
```

All actions are protected. It is not possible to execute an action without being logged in with the correct role. But this is not enough. The actions “Download New SCS” and “Download Transactions” generate a file. This file is located in a public folder where everyone has access to all files without being logged in. Therefore it is possible to download the personalized SCS and transaction history of all clients.

“In addition, it is often possible to bypass authentication measures by tampering with requests and tricking the application into thinking that the user is already authenticated. This can be accomplished either by modifying the given URL parameter, by manipulating the form, or by counterfeiting sessions.“ (OWASP Testing Guide)

Bypassing authentication schema by directly request secret pages is not possible. The page are accessible but did not contain any information. The information is filled by ajax requests to specific application actions. These actions are protected as described above.

Bypassing authentication schema by inject some sql into the authentication form is not possible because all sql statements are protected against SQL injection (see the RIPS report of chapter “whitebox analytics - php programm”)

Bypassing authentication schema by predict session ids seems not to be possible. The application uses the default php session management, which generate random session ids like (kq2p6lg1oqff2ef2mr9d7d0uc6, 6btttsso7264qr2ngoc3qbba63) with 26 alphanumeric characters.

Discovery:

Scenario 1: Download your SCS or the SCS of other clients without an active session

The SCS jar downloaded by a client is personalized as it contains a secret key private for each client. But we found that a client can download the SCS jar of other client by changing the download url.

Access with HTTP:

1. browse to http://<IP-Address>/banana_bank/exe
2. view the file
3. download any of the scs

Access with HTTPS:

On HTTPS the directory listing is not possible. But this is not a problem, because the filename is like “SCS<account number>.jar”. So you could simple guess the account number.

1. browse to https://<IP-Address>/banana_bank/exe/SCS<account number>.jar

Scenario 2: Download your transaction history or the history of other clients without an active session

We found that anyone can download the transaction history of any client by directly accessing the download url.

Access with HTTP:

1. browse to http://<IP-Address>/banana_bank/downloads
2. view the file
3. download any of the transaction history

Access with HTTPS:

On HTTPS the directory listing is not possible. But this is not a problem, because the filename is like “<account number>.pdf”. So you could simple guess the account number.

1. browse to https://<IP-Address>/banana_bank/downloads/<account number>.pdf

Likelihood:

It is very easy and obvious to test for this vulnerability. You even did not need to use any tools and need any technical skill.

Impact:

An attacker could have access to the personalized scs, which is not that problematic because he could not do an action without log in with the target account.
Downloading the transaction history of the client is quite interesting for an attacker.

Recommendation:

Do not use a public folder to prove the file. They should be a direct response on a protected action.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

We observed that there is no vulnerability. We use a frontcontroller (framework/_core/front.php) which gets all requests to our application. The controller determinants if the user is logged in and which role he has.

```
$userID = \phpsec\Session::getUserIDFromSessionID($sessionID);
$role = \phpsec\ExtendedUser::getType($userID);
```
(line 79f)

After that he checks if the role could have access to the url.

```
if(!$this->isRoleAdmittedForRoute($Request,$role))
```
(line 83)

If the role has no access, the user is redirected to a default url of his role.

If a role has access to a specific is configured in the routes.php (framework/config/routes.php). This is an example of the access rights of an employee:

```
FrontController::$AccessRights["employee"] = array( "logout",
                                                    "employees/listCustomers",
                                                    "employees/showCustomer",
                                                    "employees/approveUser",
                                                    "employees/approveTransaction");
```

Actions like downloading the personalized SCS or Transaction history are not vulnerable in our application because these files are a direct response to the requested action. They are not available in the public folder and are deleted right after sending them to the client.

Bypassing authentication schema by directly request secret pages is not possible because the pages are protected as described above.

Bypassing authentication schema by inject some SQL into the authentication form is not possible because all sql statements are protected against SQL injection (see the RIPS report of chapter “whitebox analytics - php programm”)

Bypassing authentication schema by predict session ids seems not to be possible. The application uses the session management of the owasp phpsec framework., which generate random session ids with 128 alphanumeric characters.

Test remember password functionality (OTG-AUTHN-005)

The remember password functionality is implemented in the browser. If the function is available and activated depends on user (client-site).

Our Application:

We also have a server-site "remember me" function which extends the time for auto logout from 10 mins to 1 week.

Testing for Browser cache weakness (OTG-AUTHN-006)

Other Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

The application uses static html-file which are filled with information by using AJAX. The html-files are cached but not the information retrieved by AJAX. The AJAX response has the following headers:

```
Cache-Control: no-store, no-cache, must-revalidate, post-
check=0, pre-check=0
Pragma: no-cache
```

The Application is not vulnerable.

Our Application:

Likelihood:	MEDIUM	Impact:	HIGH	Risk:	HIGH

Observation:

By looking at the source code we found out that all secret pages are vulnerable.

Discovery:

1. login as an admin or client
2. logout
3. click “back” button of your browser
4. the content of the last page before the logout is displayed

Likelihood:

The vulnerability could be abused without additional technical skill. But the attacker needs access to the victims browser and the cache of the browser needs to be filled.

Impact:

If the bank is accessed by an computer we multiple persons has access to. An attacker could get sensitive information by having a look at the history and access cached pages.

Recommendation:

To protect the secret pages we need to add the following header attributes:

```
header("Cache-Control: no-store, no-cache, must-revalidate, no-transform, max-age=0, post-check=0, pre-check=0");
header("Pragma: no-cache");
```

Testing for Weak password policy (OTG-AUTHN-007)

Other Application:

Likelihood:	HIGH	Impact:	HIGH	Risk:	HIGH

Observation:

The application uses a custom password checking function. A password is only accepted if it has at least 6 characters and contains at least one uppercase character, one lowercase character and one number.

```php
function check_pass($pass) {

    $uppercase = preg_match('@[A-Z]@', $pass);
    $lowercase = preg_match('@[a-z]@', $pass);
    $number    = preg_match('@[0-9]@', $pass);

    if(!$uppercase || !$lowercase || !$number || strlen($pass) < 6)
        return false;
    else
        return true;
}
```

(aux_func.php, lines 286 to 296)

The application does not have a password history, does not force cyclic password changes and brute force attacks are possible since there is no locking mechanism (see Weak lock out mechanism (OTG-AUTHN-003)). There is no logging of failed login attempts so brute force attacks cannot be detected.

Likelihood:

An attacker needs medium technical skills. He needs to know how to use a brute forcing tool or a programming language. Since the attacker has no time pressure in brute forcing the password and only needs the username of the victim this attack is very likely.

Impact:

If an attacker manages to brute force the password, the whole account is compromised. On the other hand an attacker cannot change the email address of an account, this allows the victim to reset his password and lock out the attacker. The Problem is that victim may not realize that the password is compromised since the webpage doesn't print a last login. If the attacker wants to transfer money, he needs either the TAN list which is password protected and only available via email or the SCS-PIN (which is only 5 digits long) depending on the type of account.

Recommendation:

Implement a stronger password policy and limit the number of failed login tries.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

We are using the OWASP phpsec framework for user management. The framework enforce a relatively complex password. It calculates a value between 0 and 1, where 1 is the strongest. When calculating the value the following factors are used:

- entropy
- are characters ordered in lexicographical order
- are characters next to each other on the keyboard
- is it a date or does it contains a date
- is it a phone number or does it contains a phone number
- does it contains duplicated words
- does it contains a number, lowercase character, uppercase character or a special character

A password is accepted if its score is high enough. The characteristics which are used for the calculation already cover a wide range of possible weaknesses so users are protected from using weak passwords.

The application does not have a password history and also does not force cyclic password changes. This isn't an optimal situation but the prevention of brute force attacks mitigates the risk dramatically.

Testing for weak password change or reset functionalities (OTG-AUTHN-009)

Other Application:

Likelihood:	NA	Impact:	NA	Risk:	LOW

Observation:

The application provides only a forgot password functionality.

The reset password functionality is based on providing the email address of the account. Next a security token is sent to this address. The original password is not changed until a new password is set using the security token. This prevents the user against denial-of-service attacks, because the current password will still be valid until the reset of the password is confirmed.

The security token is created by the following formula:

token = sha1(openssl_random_pseudo_bytes(20)).

The generated token can be regarded as safe, because of the following facts:

1. the openssl_random_pseudo_bytes function is a good pseudorandom number generator so it is really hard to predict its output

2. the token is an alphanumeric string with 20 characters and is only 2 hours valid therefore its hard to brute force the token in that time

The security of this whole process relies on the security of that email address. The application is vulnerable if an attacker manages to intercept the email or log into the email account.

Recommendation:

To improve the security of the token, you should reduce the expire time, increase its length by using another hashing function like sha512 and use a bigger input for the hash function.

Our Application:

Likelihood:	MEDIUM	Impact:	HIGH	Risk:	HIGH

Observation:

Our application provides a forgot and change password functionality.

The forgot password function is similar to the one in the other application. Instead of using the email address we use the username. Next the user will receive an email with a URL that contains a security token. Using this URL the user can login without providing username and password. The user will be redirected to the change password page. Since at this point the user has a valid session he can use other functionality of the application and is not bound to only change his password. The old password is valid as long as no new one is provided.

The security token is created by the following formula:
- randstr($length) = substr(bin2hex(openssl_random_pseudo_bytes($length)), 0, $length)
- token = hash(sha512,randstr(128))

The generated token can be regarded as safe, because of the following facts:

1. the openssl_random_pseudo_bytes function is a good pseudorandom number generator so it is really hard to predict its output
2. the token is an alphanumeric string with 64 characters and is only valid for 15 minutes therefore it's not possible to brute force the token within that time

Our application provides a change password functionality that can be used when a user is logged in. It is possible to change the password without providing the old password. This is a bad because if an attacker manages to take control of the session for example via CSRF, session prediction or a if user doesn't lock his screen while leaving his computer, the attacker can change the password.

Likelihood:

The application is protected against CSRF so an attacker can't attack the form. We use the OWASP phpsec framework session management which uses a 128 character session token so session prediction is not feasible. The only problem is if a user doesn't lock his screen or forgets to log out and an attacker can access the victims computer which is quite unlikely.

Impact:

An attacker can change the password of the account and can therefore lock out a victim. However the email address provided while registering the account cannot be changed so an attacker would need to compromise that account as well.

Recommendation:

To improve the security of password change process we should request the old password.

Authorization Testing (4.6)

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

Other Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

Since the application only uses constant file includes, it is not vulnerable.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

Files are loaded(/included) dynamically only at one place - the front controller (front.php, line 176). After checking the source code of the controller be could say, that the controller is not vulnerable, because the filenames which are loaded are constants. An user requested a route of the application. To each of these routes a constant filepath is associated, where the application finds the corresponding controller (see routes.php). Therefore the application is not vulnerable.

Testing for bypassing authorization schema (OTG-AUTHZ-002)

Other Application:

Likelihood:	HIGH	Impact:	MIDIUM	Risk:	HIGH

Observation:

Since you can download SCS and the transaction history even if you are not authenticated (see Testing for bypassing authentication schema (OTG-AUTHN-004)), the application is vulnerable.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

Since the frontcontroller checks for every route if a user needs to be logged in and if he has the correct role (see Testing for bypassing authentication schema (OTG-AUTHN-004)), the application is not vulnerable.

Testing for Privilege Escalation (OTG-AUTHZ-003)

Other Application:

Likelihood:	HIGH	Impact:	MIDIUM	Risk:	HIGH

Observation:

Since you can download SCS and the transaction history of any other client (see Testing for bypassing authentication schema (OTG-AUTHN-004)), the application is vulnerable against horizontal escalation.

The application is also vulnerable against vertical escalation because the employee also could access the SCS. But this isn't that dramatically, because an employee didn't has the rights to transfer money.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

Since the frontcontroller checks for every route if a user needs to be logged in and if he has the correct role (see Testing for bypassing authentication schema (OTG-AUTHN-004)), the application is not vulnerable.

Testing for Insecure Direct Object References (OTG-AUTHZ-004)

Other Application:

Likelihood:	HIGH	Impact:	MEDIUM	Risk:	HIGH

Observation:

We observed that it is not possible to retrieve database records of other users (like access the account details) or perform operations for other users (like transfer money from a different account), because of the following points:

- The user has no influence on decisive parameters like the email address which is for example used to retrieve the account details or create transaction. This would be necessary since the email is used in the 'from' field of the transaction.
- The application is protected against SQL injection by sanitizing inputs.

The transaction process is distributed into two parts.

1. At user login, the PDF file containing the transaction history is created in the downloads directory.
2. the user can download the PDF by click on the "Download Transaction" button/link

So the problem is that the operation to create the object (transaction history pdf) is protected but not object itself.

There is a possibility that the retrieval of file system resources is possible by exploiting a command execution vulnerability (as described in the section white box analysis using RIPS) and using the public download folder.

The application is not vulnerable against accessing application functionality since all actions are protected against bypassing authorization.

Likelihood / Impact / Recommandation:

Since the core problem of this section and the “Testing for bypassing authentication schema (OTG-AUTHN-004)” section is the same, please see section OTG-AUTHN-004.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

We observed that our application is not vulnerable.
It is not possible to retrieve database records of other users or perform operations for other users, because of the following points:

- The user has no influence on decisive parameters like the email address which is for example used to retrieve the account details or change password.
- The application is protected against SQL injection by sanitizing inputs.

Retrieving file system resources and accessing application functionality is also not possible.

Session Management Testing (4.7)

Testing for Bypassing Session Management Schema (OTG-SESS-001)

Other Application:

Likelihood:	MEDIUM	Impact:	HIGH	Risk:	HIGH

Observation:

1. We observed that the site uses a cookie named "PHPSESSID"
2. The cookie value remains same throughout the session and does not change during any action.
3. **Cookie Collection:** some cookie values are:
    a. nhm1lde1sk541m0stv7e438ou6
    b. 6dmg5p751hg2epn81cdphq9db6
    c. dbj8hh8k1jrv7jijr8n3h0nd20
    d. 2nqnjj0so2iuu2ousfual1fu03
4. **Randomness and predictability:** The cookie value is 26 characters long and is combination of lowercase alphabets and numerical values. We observed the the sessionId value is highly random and unique.
5. **Cookie reverse engineering:**
    a. The cookie is generated by php function session_regenerate_id()
    b. Secure Flag: The secure flag is not set, therefore the cookie can transferred over unencrypted network and hacked my man in the middle.

```
$res_arr = login_client_db($email, $pass);
if ($res_arr['status'] == false)
    return error($res_arr['err_message']);

session_start();
session_regenerate_id();
$_SESSION['email'] = $email;
```

Discovery:

1. Using Firebug, go to Resources => Cookies.
2. Look at the secure attribute of cookie PHPSESSID. This is not set.
3. The bank allow access over http.
4. Cookie can be transferred over http.

Likelihood:

An attacker needs medium technical skills. The attacker should know about any of the intercepting tool and he can steal the sessionId.

Impact:

An attacker could intercept the request and use the cookie to impersonate the client. At this point attacker can do everything that a client can do. Moreover if someone is able to intercept an admin or employee session that would make the whole application vulnerable.

Recommendation:

Cookies with Sensitive data like SESSIONID should be enforced to transfer over secure network only.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

1. Our application uses a cookie named "SESSIONID".
2. **Cookie Collection:** Some of the cookie examples are:
    a. aed3982fa6bd26fd122b044fd8cf3b5b4c2956012254a978543c81a8a10c59f26fdd ead776cbac7d1429b354fa987b4847fa32058a483e1d293cf4e8077f4a59
    b. ebd493335b8afe8796e6a7f1c028d1f7f635b492942e291a33a382caa176caf8ea9 7cc29fa1d78ee326a92ed86a666585062b2476e8cd226f0961705deff52b0
    c. 53d0a2092d3353c66e78dee12daaa6a6762fa9e2e8a705eb05fb4630d7bee01810 31daa8695c497292bb4795e9e4939f6ca21a12ed8502afd96f64939e1c6fc9
3. **Randomness and predictability:** Its is 128 char, highly random and unpredictable.
4. **Cookie Reverse Engineering:**
    a. SessionId is set by phpsec framework using openssl_random_pseudo_bytes. The openssl_random_pseudo_bytes function is a good pseudorandom number generator so it is really hard to predict its output.
    b. The cookies secure param is not set but since the application itself guarantees https only, the cookies cannot be transmitted over non secure networks.

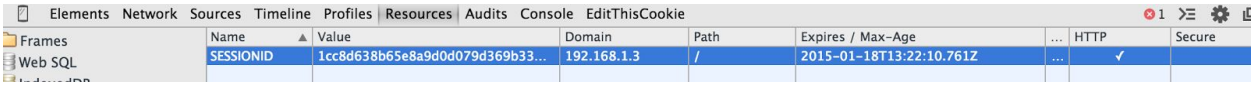
login function:

```
$this->userID = $userID;
$this->session = randstr(128); //generate a new random string for the session ID of length

$time = time(); //get the current time.
SQL("INSERT INTO SESSION (`SESSION_ID`, `DATE_CREATED`, `LAST_ACTIVITY`, `USERID`) VALUES
```

randstr function:

```
/* Use `openssl_random_psuedo_bytes` if available; PHP5 >= 5.3.0 */
if (function_exists("openssl_random_pseudo_bytes"))
{
    return substr(bin2hex(openssl_random_pseudo_bytes($length)), 0, $length);
}
```

Cookie attributes:



Testing for Cookie Attribute (OTG-SESS-002)

Other Application:

Likelihood:	MEDIUM	Impact:	HIGH	Risk:	HIGH

Observation:

We observed that the only cookie set is PHPSESSID with the following attributes.

1. Secure: False
2. HttpOnly: True
3. Domain: <IP>
4. Path: /
5. Expiry: Session

Since the secure flag is false, the cookie can be transferred over unencrypted network. Also the path is set to "/" so the cookie will be sent with to other apps also that are deployed on the server under "/".

Discovery:

1. Using Firebug, go to Resources => Cookies.

Likelihood:

An attacker needs medium technical skills. The attacker can use any intercepting tool to get the value of an active session if secure flag is not set. Also if path is "/" the other app will get all the active sessionIds. If the attacker has access to the other app deployed on same server under "/", he can get all active sessionIds.

Impact:

If a session cookie is hacked by attacker, he can impersonate client/employee.

Recommendation:

Cookies with sensitive data should be marked as secure. Path variable should refer to the exact app it is desired for.

Our Application:

Likelihood:	LOW	Impact:	HIGH	Risk:	MEDIUM

Observation:

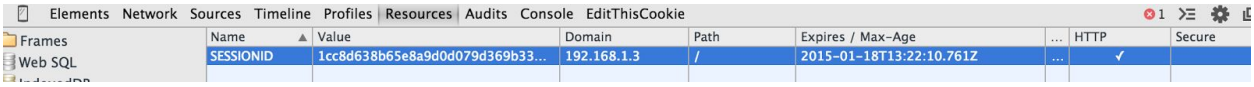
In our application the only cookie value set is SESSIONID with the following attributes.

1. Secure: False
2. HttpOnly: True
3. Domain: <IP>
4. Path: /
5. Expiry: 1 Week

Since the secure flag is false, the cookie can be transferred over unencrypted network. But this vulnerability can not be exploited in our application since "https" is enforced. Also the path is set to "/" so the cookie will be sent with to other apps also that are deployed on the server under "/".

Discovery:

Using Firebug, go to Resources => Cookies.



Likelihood:

If the attacker has access to the other app deployed on same server under "/", he can get all active sessionIds.

Impact:

If a session cookie is hacked by attacker, he can impersonate client/employee.

Recommendation:

Path variable should refer to the exact app it is desired for.

Testing for Session Fixation (OTG-SESS-003)

Other Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

We observed that the value of PHPSESSID changes on every login.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

We observed that the value of SESSIONID changes on every login.

Testing for Exposed Session Variable (OTG-SESS-004)

Other Application:

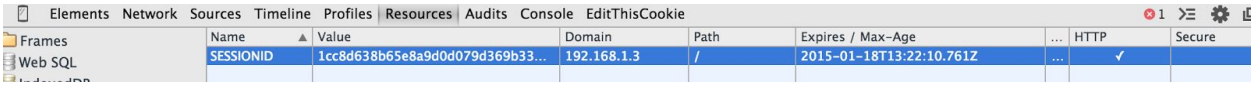
Likelihood:	MEDIUM	Impact:	HIGH	Risk:	HIGH

Observation:

We found that PHPSESSID can be transferred over unencrypted network since the application does not enforce encryption. We also observed that switching the url from "https" to "http" at any point after login, does not change the sessionId. Thus the same sessionId is being used for both https and http.

Discovery:

1. Login to site using https
2. Note the SESSIONID Cookie value using Firebug



3. Change the url from https to http
4. Make a request to server for download transaction.
5. The same sessionId cookie is sent to the server.

Likelihood:

An attacker needs medium technical skills. The attacker should know about any of the intercepting tool and he can steal the sessionId

Impact:

If a session cookie is hacked by attacker, he can impersonate client/employee.

Recommendation:

1. Either the application should be enforced to use encryption.
2. Secure flag should for session cookie should be set to true.
3. If the application want to use both "http" and "https", the sessionId cookie value should be changed on change of url from "https" to "http".

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

In our application the encryption is both default and enforced, the cookies will never be exposed.

Testing for Cross Site Request Forgery (OTG-SESS-005)

Other Application:

Likelihood:	MEDIUM	Impact:	HIGH	Risk:	HIGH

Observation:

We observed that the banana bank has CSRF Vulnerability. An attacker can make requests to the bank server using the session of an active user if the user clicks on the prompted link.

Discovery:

1. We hosted an attacker site on a domain different from the domain of banana bank.
2. We created two pages attacker1.php and attacker2.php.
3. attacker1.php makes request for approving a user with initial balance of 100000. If an employee clicks on this link the user get approved without knowledge of employee.

```
<html>
<head>
</head>
<body>
      <h1>CSRF ATTACK </h1>
        <form name="xForm" method="POST" action="http://192.168.1.3/banana_bank/php/serve_requests.php">
                <input name="action" value="approve_user" />
                <input name="email" value="test3@test3.com" />
                <input name="init_balance" value="1000" />
        </form>
        <script type="text/javascript">document.xForm.submit();</script>
        </iframe>
</body>
```

4. attacker2.php makes requests for a new transaction provided the tan number is known. If the attacker has the list if whole tan numbers, this script can be modified to try all tan numbers until success. If a client clicks on this link money get deducted from client's account without his knowledge.

```
<body>
        <h1>CSRF ATTACK 2</h1>

<form name="xForm" method="POST" action="http://192.168.1.3/banana_bank/php/serve_requests.php">
                <input name="action" value="set_trans_form" />
                <input name="account_num_dest" value="121" />
                <input name="amount" value="100" />
                <input name="tancode_value" id="tanNo" value="yu54a94f023f03e" />
                <input name="description" value="test" />
        </form>
        <script type="text/javascript">
                document.xForm.submit();</script>
        </iframe>
```

5. Further the white box analysis of code revealed that the banana bank does not have csrf protection for any of the forms.

Likelihood:

For this attack, the attacker should have basic knowledge of html for form submission. Also the victim must be enforced to click on the attacker's link either by mail or by some other means.

Impact:

An attacker can take any action of employee like approving user, approving transactions etc. Similarly an attacker can deduct any amount of money from client account and can perform other actions on behalf of client.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

Our application is safe from csrf attacks. All the forms after login have a hidden field "_csrf" that is checked on server side for each POST request. The value of csrf token is changed once it is used.

Testing for Logout functionality (OTG-SESS-006)

Other Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

A logout button is present on all pages of the site. The button is quickly identifiable. Also we verified that after logout the session is terminated on server side also. The user is not able to access any page (using the old sessionId) that require authentication after logout.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

A logout button is present on all pages of the site. The button is quickly identifiable. Also we verified that after logout the session is terminated on server side also. The user is not able to access any page (using the old sessionId) that require authentication after logout.

Test Session Timeout(OTG-SESS-007)

Other Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

We observed that the session timeout exists in the application. The session timeout is 10 minutes.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

Session timeout exists in our application also with a value of 10 minutes.

Test Session Puzzling(OTG-SESS-008)

Other Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

The analysis of code revealed that the sessionId is only set on login. There is no other context or entry point in the system that sets sessionId. The pages like forgot password and registration do not set any parameter in cookie. All the information of a user is shown only based on sessionId.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation

In our application also the sessionId is only set on login and the pages like forgot password and registration do not set any parameter in cookie.

Data Validation Testing (4.8)

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

Other Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

There were no reflected XSS vulnerabilities. This is due to the fact that the login is handled in an AJAX request. In case of wrong login credentials as we use them, the input is not touched and therefore not interpreted as java code.

Discovery:

1. Create a new transaction
2. insert '<script>alert();</script>' in the Destination field
3. submit the transaction
4. script is not executed

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

We did not observe any reflected XSS vulnerability

Discovery:

1. Insert '<script>alert();</script> as username
2. Try to login without a password
3. The script is still in the username field but is not executed. Looking at the HTML source shows that all characters that have a special meaning in HTML have been converted to their HTML-entities.

The sanitization in the source code is executed in the Start method of DefaultController which is an abstract class from which all others are derived.

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

Other Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

Banana bank is not vulnerable against stored XSS attacks. A detailed explanation for this can be found in the RIPS report (see chapter "Whitebox analytics - php programm").

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

Since all input is sanitized before being processed in the application (see OTG-INPVAL-001), stored XSS attacks are prevented.

Testing for HTTP Verb Tampering (OTG-INPVAL-003)

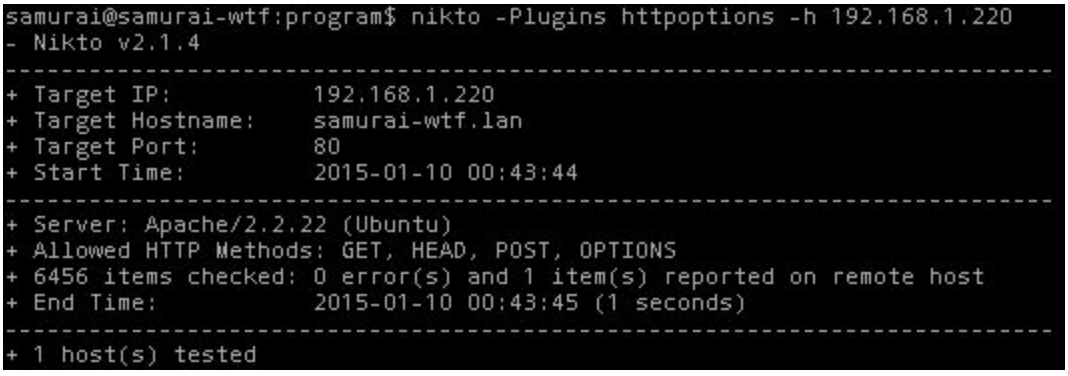
Other Application:

Likelihood:	LOW	Impact:	LOW	Risk:	LOW

Observation:

The server supports the HTTP methods OPTIONS, HEAD, POST and GET. According to a paper “Bypassing Web Authentication and Authorization with HTTP verb”.

Discovery:

1. Execute nikto with the command ‘`nikto -Plugin httpoptions -h <IP-ADDRES>`’
2. 

According to the aforementioned paper HEAD requests can be used to circumvent access restrictions.

Likelihood:

An attacker needs deep knowledge of recent publications in the security sector

Impact:

Since the application doesn’t use role restriction based on HTTP method type there is only low impact.

Recommendation:

Deactivate unneeded HTTP methods and start using access restriction like .htaccess files.

Our Application:

Likelihood:	LOW	Impact:	LOW	Risk:	LOW

Observation:

Our application has the same vulnerability as the banana bank since the same HTTP methods are activated.

Testing for HTTP Parameter pollution (OTG-INPVAL-004)

Other Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

As stated in the OWASP testing guide
https://www.owasp.org/index.php/Testing_for_HTTP_Parameter_pollution_%28OTG-INPVAL-004%29#Expected_Behavior_by_Application_Server
in the combination Apache/PHP only the last occurrence of a parameter is passed on to the application. Therefore discrepancies of checking the first occurrence of a parameter but using the last cannot happen.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

Since our application uses the same combination of Apache/PHP as banana bank the above statement holds true for our application as well.

Testing for SQL Injection (OTG-INPVAL-005)

Other Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

In the RIPS analysis (see RIPS Report in chapter “Whitebox analytics - php program) we found out that all input is sanitized before being used in SQL statements so the application is save against SQL injections.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

We use the phpsec SQL function that internally uses prepared statements. We made sure the usage of prepared statements is done correctly in the file 'phpsec/libs/db/adapter/base.php' in the function 'SQL'(lines 141 to 194). We also added the handling of calls to stored procedures since that was missing in the framework. Since the framework handles prepared statements correct, the application is not vulnerable to SQL injection.

Testing for Code Injection (OTG-INPVAL-012)

Other Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

We did not find any code injection vulnerability in the banana bank code base. However a related vulnerability that is sometimes confused with code injection is command (OTG-INPVAL-013) was found.

Discovery:

We scanned the source code for typical functions that can lead to code injections via find and grep. The function names we scanned for are: "system, exec, shell_exec, popen" however we found a source that states that RIPS is testing the source for a far bigger list of dangerous functions so no new insights could be gained from this scan compared to the RIPS report.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

The RIPS report shows that there are no vulnerable points for Code Injection.

Testing for Local File Inclusion and Testing for Remote File Inclusion (also OTG-INPVAL-012)

Other Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

There were no vulnerable file inclusions in the banana bank source code, neither of remote nor of local files.

Discovery:

We scanned the source code using find and grep as in '`find path_to_source -name *.php -exec grep -n -H --color -e "require" -e "include" {} +`' which showed us all the calls to "require, require_once, include and include_once". None of these calls included any variables so there cannot be any file inclusion.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

There were no vulnerable file inclusions to be found in the source code of our application.

Discovery:

We used the command '`find phpsec/  -name *.php -exec grep -n -H --color -e 'require.*\$' -e 'include.*\$' {} +`' to find all possible occurrences of dangerous file inclusion. This left us with a list of 12 possibly dangerous call sites. Manually checking these 12 calls to either "require_once" or "include" showed that the variables used are not controlled by the user and therefore safe.

Testing for Command Injection (OTG-INPVAL-013)

Other Application:

Likelihood:	LOW	Impact:	HIGH	Risk:	MEDIUM

Observation:

We found a command injection vulnerability that allows the execution of commands with root privileges.

Discovery:

We scanned the source code with RIPS and found 3 candidates for command execution, one of those is exploitable.

1. Log in as a client and go to new transactions page

2. Send requests to server via OWASP Zed Attack Proxy and break at requests
3. Upload a file
4. In ZAP change the filename to e.g. “| touch x” and Content-Type to “text/plain”
5. unblock requests and send it to the server
6. go to “http://<IP-ADDRESS>/banana_bank/php” and check that the file x has been created

Likelihood:

This attack requires advanced knowledge of shell commands and the usage of a proxy so the likelihood is low.

Impact:

Since this vulnerability allows the execution of commands even as root (sudo does not require a password) this is very severe. We didn’t find a way to use slashes in the commands but a creative attacker might find a way around this limitation.

Recommendation:

Sanitize user input before using it in shell commands and avoid execution of OS commands in general.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

Our application is not vulnerable against command injection.

Discovery:

There is only one place where an external command is called via ‘shell_exec’ (framework/control/customers/CreateTransactionController.php, line 311) and no part of the command is directly controller by the user.

Testing for Buffer overflow, Testing for Heap overflow ,Testing for Stack overflow and Testing for Format string (OTG-INPVAL-014)

Other Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

We did not find any buffer overflows in the banana bank application

Discovery:

Since it is a PHP web application the only place to find this kind of vulnerability is the C binary used on the server. A detailed analysis of it can be found in the section “Reverse Engineering - C++ Program”

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

We did not test our C++ application regarding overflows.

Testing for incubated vulnerabilities (OTG-INPVAL-015)

Other Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

We already covered some possible vulnerabilities that belong to the class of incubated vulnerabilities. Like stored XSS. none of them were successful so we deduce that there are no incubated vulnerabilities. Another vulnerability that falls into this category and which we could find for banana bank is file creation on the server using command injection.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

We also couldn't find any incubated vulnerability in our application as stored XSS was not working.

Testing for HTTP Splitting/Smuggling (OTG-INPVAL-016)

Other Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

We found no vulnerabilities that are affected by HTTP splitting nor smuggling

Discovery:

Using RIPS we found no possible vulnerabilities of HTTP splitting where some kind of user input has influence on the HTTP response. Likelihood:

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

We found no vulnerabilities that are affected by HTTP splitting nor smuggling

Discovery:

Using RIPS we found 4 possible vulnerabilities of HTTP splitting where some kind of user input has influence on the HTTP response. Two of those discoveries were in example controller files delivered with the library. The other two were related to forgotten password and temporary password functionality. It turned out that both instances don't use unsanitized user input.

Error Handling (4.9)

Analysis of Error Codes (OTG-ERR-001) and Analysis of Stack Traces (OTG-ERR-002)

Other Application:

Likelihood:	HIGH	Impact:	LOW	Risk:	LOW

Observation:

We observed that the application uses default error pages for web server errors. The page contains sensitive information about the host system (used web server + version, host os).
The application uses AJAX quests. The responses of this request contain the information which version of php is used.

The application itself proved many different error messages. These messages doesn't prove any information about server paths, installed libraries or application version. If a database error occurs the application does not report the database error message it the user but it reports a customized application error message.

Discovery:

1. browse to https://<IP-Address>/banana_bank/download

2. browse to https://<IP-Address>/banana_bank/downloads

3. browse to https://<IP-Address>/banana_bank/php/serve_request.php

```
HTTP/1.1 200 OK
Date: Mon, 12 Jan 2015 13:54:01 GMT
Server: Apache/2.2.22 (Ubuntu)
X-Powered-By: PHP/5.3.10-1ubuntu3.15
Content-Length: 59
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: application/json

{"status":"false","message":"Accepting only POST requests"}
```

Likelihood:

It is easy to gain some information about the server infrastructure.

Impact:

An attacker could use the information about server infrastructure to search for known vulnerabilities of the software. The impact depends how vulnerable the server software is.

Recommendation:

Do not use default error messages and customize the server response.

Our Application:

Likelihood:	HIGH	Impact:	LOW	Risk:	LOW

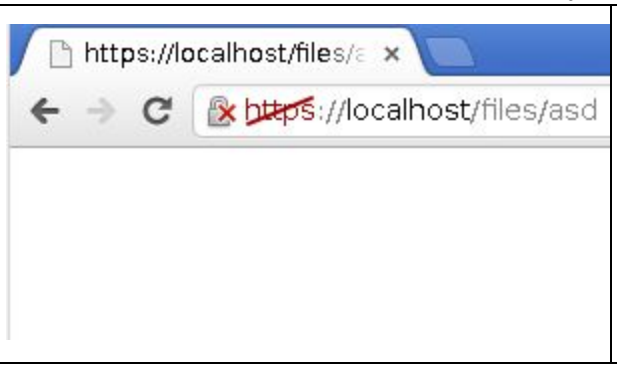
Observation:

Our application doesn't use default server error messages if you try to access non existing folders/file. In case of requesting a url like "https://<ip-address>/files/…" you will be get an empty page. In all other cases you will be redirected to "https://<ip-address>/login". So the page itself does not contain sensitive data but its response attributes. It contains the following information: server os + version, used web server + version, used web technology (php) + version.

The application itself proved many different error messages. These messages doesn't prove any information about server paths, installed libraries or application version. If a database error occurs the application does not report the database error message but it crashes and show an empty page.

Discovery:

1. browse to https://<IP-Address>/files/xyz




```
HTTP/1.1 200 OK
Date: Mon, 12 Jan 2015 14:29:13 GMT
Server: Apache/2.2.22 (Ubuntu)
X-Powered-By: PHP/5.3.10-1ubuntu3.7
Vary: Accept-Encoding
Content-Length: 0
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html
```

2. browse to https://<IP-Address>/asd

```
HTTP/1.1 302 Found
Date: Mon, 12 Jan 2015 14:29:03 GMT
Server: Apache/2.2.22 (Ubuntu)
X-Powered-By: PHP/5.3.10-1ubuntu3.7
Location: https://localhost/login
Vary: Accept-Encoding
Content-Length: 0
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html
```

you will be redirected to https://<IP-Address>/login

Likelihood / Impact:

see above on “other application”

Recommandation:

Customize the server response.

Cryptography (4.10)

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

Both Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

We observed that both application should be secure. We used nmap to get the following information:

- length of rsa key is 2048 which is secure
- all possible ssl/tls ciphers

NMAP categories all supported ciphers as strong, so no vulnerability exists.

Discovery:

1. execute nmap: nmap --script ssl-cert,ssl-enum-ciphers -p 443 <ip-address>

```
samurai@samurai-wtf:~$ nmap --script ssl-cert,ssl-enum-ciphers -p 443 192.168.56.101

Starting Nmap 6.25 ( http://nmap.org ) at 2015-01-12 17:50 CET
Nmap scan report for 192.168.56.101
Host is up (0.0013s latency).
PORT    STATE SERVICE
443/tcp open  https
| ssl-cert: Subject: commonName=ubuntu
| Issuer: commonName=ubuntu
| Public Key type: rsa
| Public Key bits: 2048
| Not valid before: 2012-05-23T14:43:42+00:00
| Not valid after:  2022-05-21T14:43:42+00:00
| MD5:   585f 5275 4290 bd6d 4a31 de0e 45ba 2ae4
|_SHA-1: b4db 28ca 21d7 d121 79c5 8c29 c657 8fd8 e256 e36a
| ssl-enum-ciphers:
|   SSLv3:
|     ciphers:
|       TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA - strong
|       TLS_DHE_RSA_WITH_AES_128_CBC_SHA - strong
|       TLS_DHE_RSA_WITH_AES_256_CBC_SHA - strong
|       TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA - strong
|       TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA - strong
|       TLS_DHE_RSA_WITH_SEED_CBC_SHA - strong
|       TLS_RSA_WITH_3DES_EDE_CBC_SHA - strong
|       TLS_RSA_WITH_AES_128_CBC_SHA - strong
|       TLS_RSA_WITH_AES_256_CBC_SHA - strong
|       TLS_RSA_WITH_CAMELLIA_128_CBC_SHA - strong
|       TLS_RSA_WITH_CAMELLIA_256_CBC_SHA - strong
|       TLS_RSA_WITH_RC4_128_SHA - strong
|       TLS_RSA_WITH_SEED_CBC_SHA - strong
|     compressors:
|       NULL
```

Testing for Sensitive information sent via unencrypted channels (OTG-CRYPST-003)

Other Application:

Likelihood:	LOW	Impact:	LOW	Risk:	LOW

Observation:

Since the whole application is accessible via HTTP all information (login data, cookies, …) could be send via an unencrypted channel. Even if the whole application uses HTTPS, it is still vulnerable, because the SCS TAN is send via unencrypted email to the user.

Discovery:

1. register new user and select SCS Application
2. approve user by employee
3. email is unencrypted and contains the scs pin

Likelihood:

An attacker needs to intercept the email communication or break into your email account. This needs high technical skills.

Impact:

If an attacker only manages to get the SCS, he couldn't to anything if he hasn't access to the bank account.

Recommendation:

Print the SCS PIN after registration like the password for tanlist pdf.

Our Application:

Likelihood:	LOW	Impact:	LOW	Risk:	LOW

Observation:

Since the whole application uses HTTPS all information is send via encrypted channel. As the other application we also send the SCS PIN via unencrypted channel, so the application is vulnerable.

Likelihood / Impact / Recommandation:

It the same as for the other application.

Business Logic Testing (4.11)

We did not manage to make some batch transaction, because they did not provide a template file. The file should look like this but it doesn't work:

```
122 2000 Trans2
km547d7b9395324
```

We alway get the following error: "Uploaded file does not comply with rules. Last line should have only TAN code or SCS token"
So there are maybe some vulnerabilities which were not found.

Test business logic data validation (OTG-BUSLOGIC-001)

Other Application:

Likelihood:	HIGH	Impact:	MEDIUM	Risk:	MEDIUM

Observation:

We observed that there is no input validation on the client site. All the inputs are validated on the server-site. The application is good protected against for example

- transferring negative amount
- use a wrong tan for transactions
- use text instead of numbers for destination and amount
- try to access non existing clients as an employee
- use text instead of numbers for initial balance while approving new clients
- and so on.

But the validation has a few gaps.

Broken SCS validation:
If the account uses SCS codes to validate transaction, we found that single transactions are not protected. The application accepts any random 20 digit string as SCS code without even using the SCS jar at all.

Create transaction where source and destination is equal:
It is possible to successfully create a single transaction where source and destination is equal. The transaction is shown in the transaction history. The balance of the account doesn't so change so it has no impact.

Burn transaction tans:
It is possible to consume tans without performing a transaction. This happens when you are submitting a single transaction where everything is correct except the destination account. The destination account should not exist.

Requested TAN fixation:
We observed that the requested tan only changes it the whole page is reloaded. Especially it is not changed if you submit a transaction with a wrong tan.

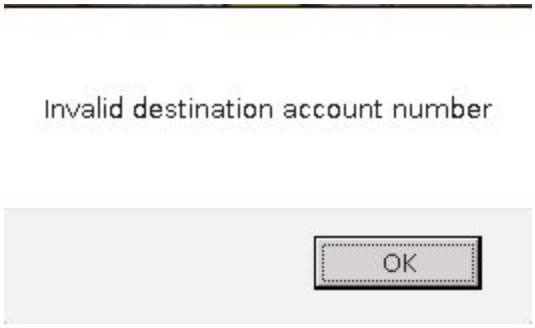
Discovery:
Examples of protection:

1. login as a client
2. create a transaction we amount is “asd” like this and submit the transcation

```
action=set_trans_form&account_num_dest=122&amount=asd&tancode_value=yi547d7b939496d&description=aasdadsd
```

3. an error message is shown:

4. create an transaction where the destination account number is a text and submit it
5. an error message is shown:



6. login as an employee
7. try to access the transaction history on an client
8. intercept the request with ZAP Proxy and delete the POST parameter “email”. The quest should look like this:

```
action=get_trans_emp
```

9. The response contain the following message:

```
{"status":"false","message":"Email not specified"}
```

Scenario 1: Broken SCS validation

1. login as a client that uses scs
2. go to page “https://<ip-address>/banana_bank/html/clientNewTransaction.html”

3. fill out the form to create a single transaction. use any 20 digit long string for the “TAN N: -1” field and submit the transaction
4. a new transaction is show in the transaction history

Scenario 2: Create transaction where source and destination is equal

1. login as a client
2. create a single transaction where source and destination is equal and submit it
3. view the transaction history. A new item is inserted and the account balance did not changed

Scenario 3: *Burn transaction tans*

1. login as a client
2. create a single transaction where the destination account number is a non existing account number. all other fields are filled correctly
3. an error message is displayed with the following message: “Destination account is not registered or approved”
4. change the destination account number to an existing account number and submit again
5. an error message will tell you that the tan is already used
6. view the transaction history. No item is inserted

Scenario 4: *Requested TAN fixation*

1. login as a client
2. submit a transaction with a invalid tan
3. the requested tan doesn’t change
4. reload the page
5. the requested tan has changed

Likelihood:

All vulnerabilities are detected without additional technical skills.

Impact:

Some of the vulnerabilities have an high impact like the broken scs validation and fixation of the requested tan. Both lead to a high security risk.
Requested tan fixation could be used to brute force a tan. The requested tan only changes if the page is reloaded. If the attacker know only one valid tan, he could reload the page until the tan is requested that he know.

Recommendation:

Validation input parameter in the front- and backend. Change the requested tan after every submission of an transaction.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

We found that the input data is only validated on the server site. It also should be validated on the client site.
We did not found any vulnerabilities.

Test Ability to forge requests (OTG-BUSLOGIC-002)

Other Application:

Likelihood:	MEDIUM	Impact:	MEDIUM	Risk:	MEDIUM

Observation:

We observed that the application is vulnerable. It is possible to break the business logic workflow by rejecting already approved transactions. Normally a transaction item shouldn't be editable or deletable after a transaction is approved.

We also tried one of the following examples but the application wasn't vulnerable:

- register the same employee / client twice
- approve the same transaction or user twice
- reject an already approved user
- and so on ...

Discovery:

1. client: create a transaction
2. employee: approve the transaction if the amount was above 10.000
3. client: the transaction is shown in the transaction history
4. employee: delete the transaction with the following POST request to server_request.php by using ZAP proxy

```
action=reject_trans&trans_id=4
```

5. client: the transaction isn't shown in the transaction history

Likelihood:

An attacker needs technical skills to do the attack because he needs to perform a csrf on an logged in employee, which is maybe not that easy.

Impact:

The impact of deleting transactions is medium. If the attacker manages to perform a transaction and he did not want the victim to know. He could delete the transaction and therefore hide his existence.

Recommendation:

Check is a transaction is already approved before deleting it.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

We did not found any vulnerability.

Test integrity checks (OTG-BUSLOGIC-003)

Other Application:

Likelihood:	MEDIUM	Impact:	LOW	Risk:	LOW

Observation:

We also observed that it is possible to store a transaction description which is longer than 100 characters via a single transaction. The frontend limits the number of characters, but you could bypass this by using ZAP.

```
<td class="body3" >Description (max. 100 characters):</td>
<td><label><textarea name="description" rows="4" id="desc" style="width:90%;" maxlength="100"></textarea></
```

The problem is that the server did not check the length of the input parameter. It directly inserts the input into the database. The database field so 120 characters long. So the database limits the maximum characters.

Discovery:

1. login as a client
2. create a transaction, submit it and intercept the request with ZAP Proxy
3. change the description to a string which is longer than 100 character and smaller than 120
4. view the now transaction with a description longer than 100 characters

Likelihood:

An attacker needs medium technical skills because he needs to know how to use ZAP Proxy.

Impact:

The impact is very low because the attacker could only store 20 additional characters.

Recommendation:

An application should not rely on non-editing controls, drop-down menus or hidden field.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

We couldn't find an vulnerability.

Test Upload of Unexpected File Types (OTG-BUSLOGIC-008)

Other Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

Banana bank checks the submitted file for the “Content-type” to protect against malicious file uploads.

Discovery:

1. Choose a binary file for uploading
2. Block the request in ZAP
3. Change Content-type to “text/plain”
4. Unblock requests and send the forged request
5. The file is rejected saying only text files are supported.

The check for this can be found in the function “upload_file” in aux_func.php line 243. It uses the $_FILE[] array provided by PHP.

Our Application:

Likelihood:	MEDIUM	Impact:	HIGH	Risk:	HIGH

Observation:

Our application reacts different on different types of files an unexpected file type always leads to an error. So we could assume that it can handle unexpected file types. However looking at the source code shows us that the PHP application relies on the C++ file to properly abort on wrong file inputs.

Discovery:

1. Choose a binary file to upload
2. Intercept request in ZAP
3. Change file type to text/binary
4. “There was an error processing on the file, please try again!” is shown

The checks for uploaded files are in the function ‘isInputValidForBatchTransaction’ in the file ‘phpsec/framework/control/customers/CreateTransactionController.php’, line 385.

Likelihood:

An attacker needs no special knowledge, only choose any file to upload. Crafting a file to really do damage takes deep knowledge of overflowing techniques so. Therefore the likelihood is medium.

Impact:

The C++ program could behave unexpectedly, even leading to buffer overflows and code execution. Therefore the impact is high.

Recommendation:

Restrict file types so only text files can be uploaded.

Client Side Testing (4.12)

Testing for DOM Based Cross Site Scripting(OTG-CLIENT-001), Testing for Javascript injection (OTG-CLIENT-002), Testing for HTML Injection (OTG-CLIENT-003)

Other Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

Analysing the javascript code of banana bank in "funcions.js", we found that application is safe from DOM Based Cross Site Scripting, Javascript Injection and HTML injection. Although the application make heavy use of javascript to load the content in browser but nowhere in the js code the application takes the user input and display on GUI. Only the data returned from the server is shown on GUI (the server escapes the data). The application is essentially safe from any such client side attack.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

Our application is also safe from these client side attacks. Our application does not make heavy use of javascript. The only javascript used is "check.js" that is only used for validating input data. The js does not display any user input on GUI.

Testing for Clickjacking(OTG-CLIENT-009)

Other Application:

Likelihood:	MEDIUM	Impact:	HIGH	Risk:	HIGH

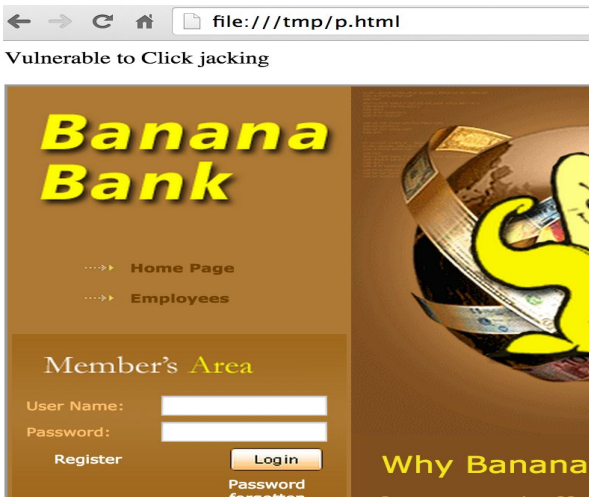
Observation:

We observed that application is vulnerable to clickjacking. The application can be opened in an iframe of some other page.

Discovery:

1. We wrote a simple html page p.html.
2. We found that the banana bank site can be opened in an iframe and thus is vulnerable to clickjacking

```
<html>
<body>
<p>Vulnerable to Click jacking</p>
<iframe src="http://192.168.1.3/banana_bank/html/index.html" width="500" height="500">
</body>
</html>
```



Likelihood:

An attacker needs knowledge of html and css to exploit clickjacking.

Impact:

With a successful attack, an attacker can make an employee to approve users, approve transaction etc without his knowledge. If in future, application take large inputs from mouse clicks at client side also, then this can further be exploited .

Recommendation:

It is a good practice, not to allow applications for banking, e-commerce to be opened in iframe.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation

It is not possible to open our application in an iframe. If the application is opened in iframe, it will detect that its not the top element and will make itself the top element of DOM. This is done with the following js code.

```
<style id="antiClickjack">body{display:none !important;}</style>
<script type="text/javascript">
        if (self === top) {
                var antiClickjack = document.getElementById("antiClickjack");
                antiClickjack.parentNode.removeChild(antiClickjack);
        } else {
                top.location = self.location;
        }
</script>
```

Test Local Storage (OTG-CLIENT-012)

Other Application:

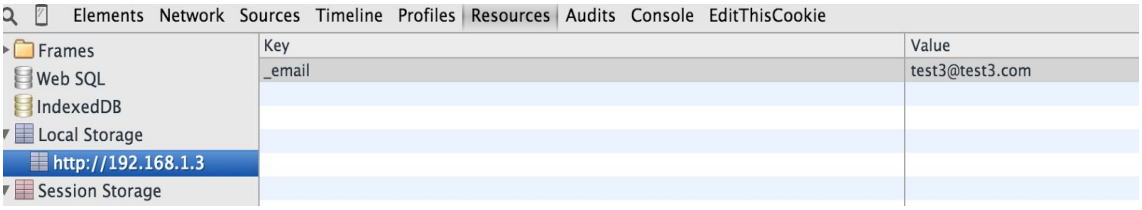
Likelihood:	LOW	Impact:	LOW	Risk:	LOW

Observation:

We observed that banana bank uses localStorage of browser to store emailIds of clients. We believe that any such sensitive information of bank should not be stored in browser area. Even if the browser is closed, the localStorage is not deleted. If the attacker manage to access the machine or have an XSS attack, then this data is exposed to the attacker.

Discovery:

1. Login with employee and view details of a client.
2. Use Chrome->Inspect Element -> Resource-> LocalStorage



3. We observed that client information was visible in localStorage.
4. Analysing the code revealed that the "emailId" is not sanitized before saving/retrieving them in localStorage that is generally considered as a good practice to protect the application from any kind of injections in the future.
5. Though the application is overall safe so this loop hole cannot be exploited.

```
localStorage.setItem('_email', text);
window.open("employeeViewClient.html", "_self");
```

```
$(function(){
        var email = localStorage.getItem('_email'
var data = "action=get_account_emp&email="+email;
```

Likelihood:

Likelihood is low since the attacker should have access to victim's machine to read the local storage or a successful xss attack can make this possible.

Impact:

Sensitive information of bank, emailId of its clients, is exposed to attacker.

Recommendation:

Bank site should not use localStorage to store sensitive information.

Our Application:

Likelihood:	NA	Impact:	NA	Risk:	NA

Observation:

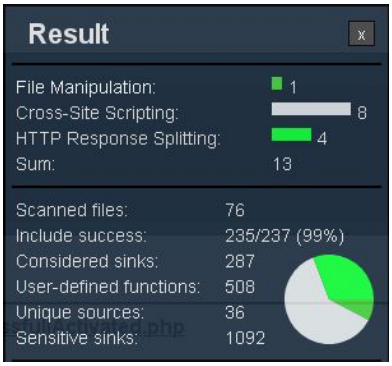
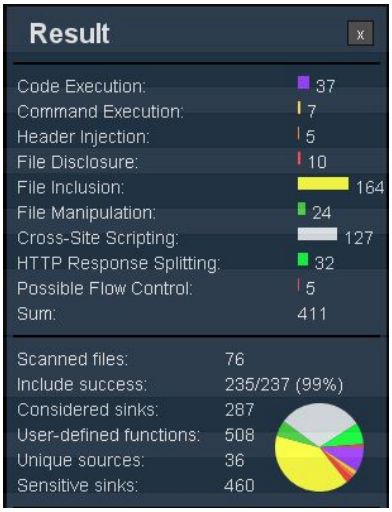
Our application does not make use of localstorage of browser so is essentially safe from this kind of vulnerability.

Whitebox Analysis - PHP Program

We use RIPS (version 0.54) for whitebox analysis of the php code of our application and the other application.

Our Application:

We analysed the source code of the framework folder and search for ‘vuln type: all’. The results for the 4 different modes are:

1. user tainted only 2. file/DB tainted 3. show secured	4. untainted
	
13 detected Vulnerabilities	411 detected Vulnerabilities

Our application is written in OOP-style. RIPS doesn’t support OOP very well so it detects only a small number of vulnerabilities in mode 1-3 and a really huge number in mode 4. Most of them are probably false positives. So we did not have a look for all vulnerabilities in detail, but we checked some samples which were all false positives.

RUN (1. user tainted only, 2. file/DB tainted, 3. show secured)

File Manipulation

RIPS found one file manipulation vulnerability in the CreateTransactionController.php. The controller uses the php function move_uploaded_file.

```
move_uploaded_file move_uploaded_file($source_path, $target_path))
```
(line 314)

The parameter $source_path and $target_path may contain unsecure filenames. The $source_path variable gets his value from the global variable $_FILES. The 'tmp_name' is randomly generated temporary name which is set by php, so it secure.

```
$source_path = $_FILES['uploadedfile']['tmp_name'];
```
(line 311)

The $target_path variable is a concatenated string of constants and the $target_name variable. The $target_name variable contains a randomly generated string, so it is also secure.

```
$target_path = __DIR__ . '/../../uploads/' . $target_name . '.txt';
```
(line 313)

```
$target_name = \phpsec\Rand::randstr(20);
```
(line 312)

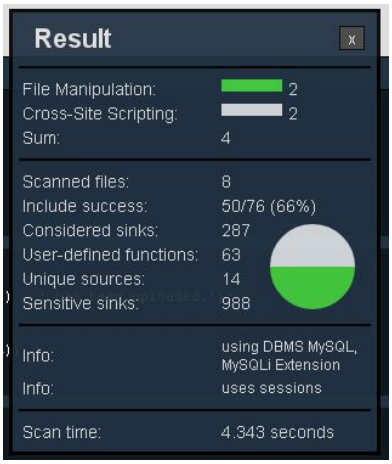
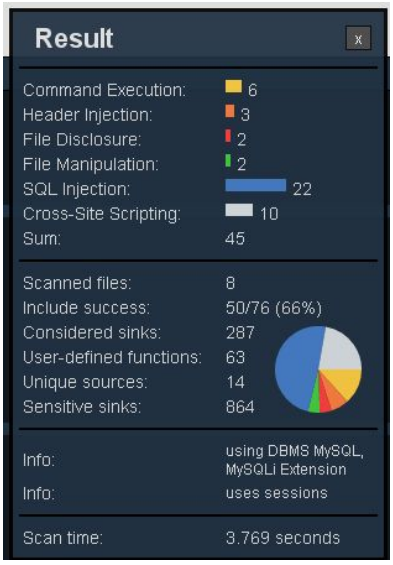
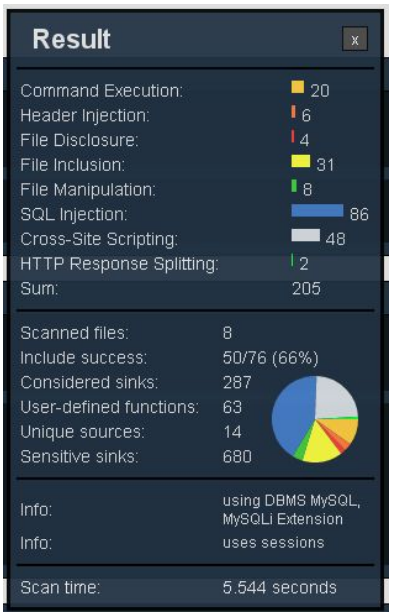
The vulnerability is a false positive.

Cross-Site Scripting

All the items regarding cross-site scripting are false positives because every user input (GET/POST) is sanitized against XSS. Every controller is based on the DefaultController (/framework/_core/base/control.php). The controller has a start function which is called after a new controller is created. The start function sanitizes the input in line 26.

Other Application:

We analysed the php-folder of the source code and searched for ‘vuln type: all’. The results for the 4 different modes are:

1. user tainted only 2. file/DB tainted	3. show secured	4. untainted
		
4 detected Vulnerabilities	45 detected Vulnerabilities	200 detected Vulnerabilities

The number of vulnerabilities is relatively high. But some of them are counted twice, so the real number of vulnerabilities is lower.

The application defines 63 different functions. Some of them are used many times by other functions:

declaration	count of calls
print_debug_message	266
error	232
sanitize_input	52
get_dbconn	46
close_dbconn	44
is_valid_session	38

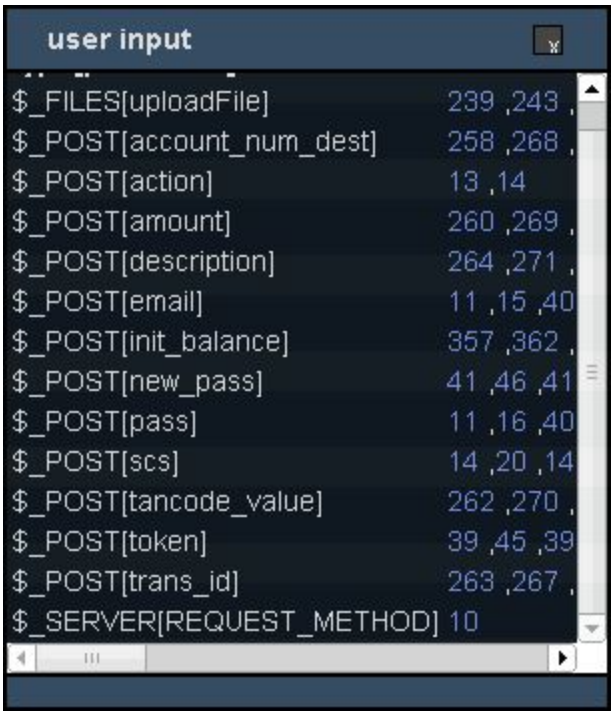
The mail_reject_trans-function is used from non other function. So maybe it is dead code.

The sanitize_input-function trims the input, removes all backslashes and escapes special characters of html. The function sanitizes input against cross site scripting and cross site request forgery.

It also helps a little bit against command and filepath injection, but it is not full protection.

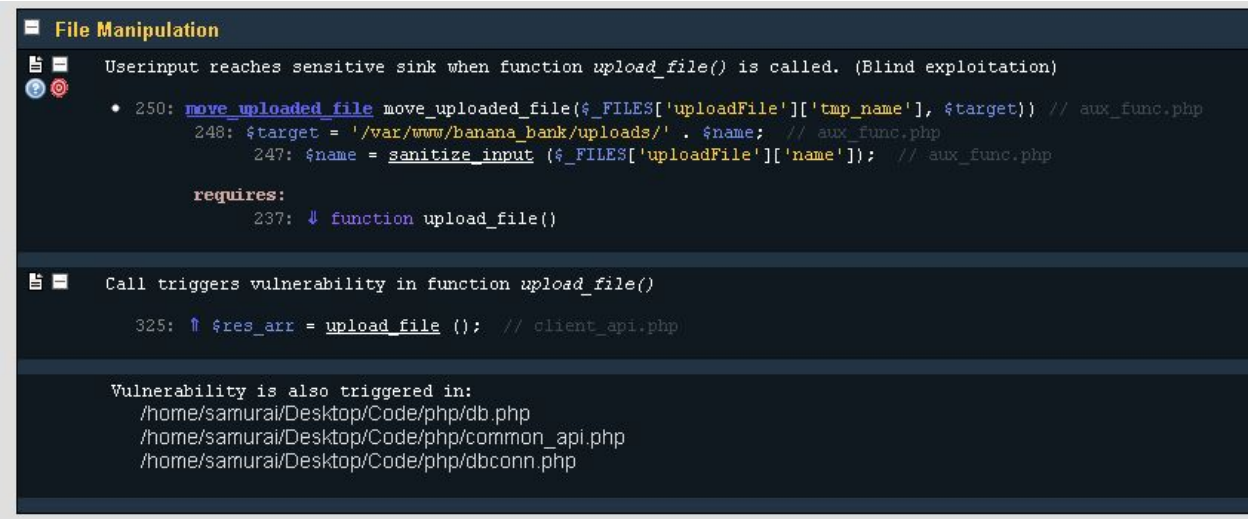
```
function sanitize_input($input) {

    $input = trim($input);
    $input = stripslashes($input);
    $input = htmlspecialchars($input);

    return $input;
}
```

The application uses one server, one file and 12 post variables. Using only post variables is good because it raises the bar a little bit against CSRF attacks.



RUN 1 (1. user tainted only and 2. file/DB tainted)

File Manipulation



There are two “File Manipulation” vulnerabilities detected in file “Code/php/employee_api.php” and “Code/php/serve_requests.php”. The upload_file()-method (line 237 of php/aux_func.php) concatenates the filename, which is given by the user request, directly at the end of the destination filepath. By changing the filename to ‘../../test.txt’ instead of ‘test.txt’ you can save the file to another location than expected. If you are using php version greater than 5.3.6 this is not a problem, because php fixed this issue. It would automatically translate ’../../test.txt’ to ‘test.txt’.

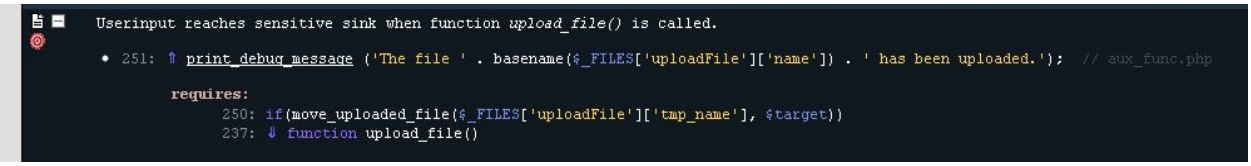
The application is not vulnerable. It is a false positive.

Cross-Site Scripting

Function print_debug_message($message) may be vulnerable for cross-site scripting because it echos the $message parameter directly to the response without sanitizing it. If a vulnerability exists or not depends on the $message parameter. If the message is a constant or is sanitized before, everything is fine.

The print_debug_message function is called by these functions: upload_file, parse_file and transfer_money.

upload_file:



The upload_file-function wants to print the name of the uploaded file. This name is already sanitized by php version > 5.3.6 and by the basename-function so no vulnerability exist.

parse_file:

The parse_file-function directly uses the $filename-parameter without sanitizing. The parse_file-function is called by the set_trans_file-function which calls the upload_file-function. The upload_file-function sanitizes the input with the sanitize_input-function, so no vulnerability exist.

Function Stack Trace:

… ⇒ set_trans_file
⇒ upload_file
⇒ sanitize_input
⇒ parse_file

transfer_money:

The transfer_money-function uses the parameters $account_num_src, $account_num_dst and $amount as concatenated input for the print_debug_message-function.

The parameters $account_num_src and $account_num_dst are used in a sql statement. If the statement gets no tuple or produces an error, the transfer_money-function will be left. So both parameters are numbers and didn't need to be sanitized.

The transfer_money-function require that the $amount is an number. So we need to have a closer look at the function stack trace. There are three scenarios.

Function Stack Trace 1:

```
user request
        ⇒ approve_trans
                ⇒ approve_trans_db
                → all values are read from the database. The database fields are numeric.
                        ⇒  transfer_money
```

Function Stack Trace 2:

```
user request
        ⇒  set_trans_form
        → checks if amount is of type float
                ⇒ set_trans_form_db
                        ⇒ transfer_money
```

Function Stack Trace 3:

```
user request
        ⇒  set_trans_file
        → amount is directly read from transaction batch file
                ⇒ set_trans_file_db
                → checks if amount ($params[$i][1]) is of type float
                        ⇒ transfer_money
```

In all szenario the amount is a number, so no vulnerability exists. It is a false positive.

RUN 2 - 3. show secured

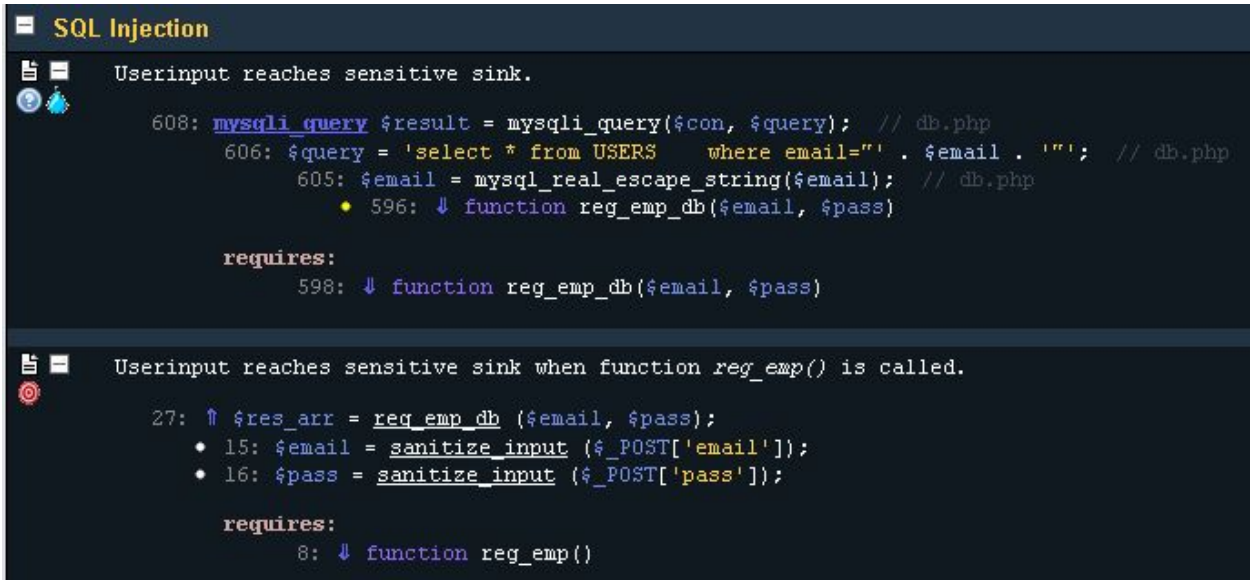
Nr	Function	serve_request.php	employee_api.php	client_api.php
SQL Injection				
1	req_emp_db	-	item 6, 7	-
2	login_emp_db	-	item 8	-
3	get_account_emp_db	-	item 9	-
4	get_trans_emp_db	-	item 10	-
5	approve_trans_db	-	item 11, 12	-
6	reject_trans_db	-	item 13	-
7	approve_user_db	-	item 14 - 18	-
8	reject_user_db	-	item 19	-
9	recover_pass_db	item 9, 10	-	-
10	change_pass_db	item 11, 12	-	-
11	req_client_db	item 13, 14	-	-
12	login_client_db	item 15, 16	-	-
Cross-Site Scripting				
1	print_debug_message	item 2	item 2	-
2	get_account_emp	item 18	item 20	-
3	get_trans_emp	item 19	item 21	-
4	req_client	item 17	-	item 3
5	error	item 1	item 1	-
Command Execution				
1	mail_tancodes	item 3	item 3	-
2	parse_file	item 8	-	item 2
3	get_trans_emp_pdf	item 20	item 22	-

File Manipulation				
1	upload_file	item 6	item 5	-
Header Injection				
1	mail_reject_account	item 4	item 4	-
2	mail_token	item 5	-	-
File Disclosure				
1	parse_file	item 7	-	item 1
Sum:		20 items	22 items	3 items

Most of the reported vulnerabilities are false positives.

SQL Injection - 1 - req_emp_db

Item 6:



The reg_emp_db-function uses its input parameter $email in a sql query.
`$query = 'select * from USERS where email="' . $email . '"';` (db.php, line 606/607)
$email is sanitized against sql injection so no vulnerability exist. It is a false positive.
`$email = mysql_real_escape_string($email);` (db.php, line 605)

Item7:

The reg_emp_db-function uses its input parameter $email and the $hash variable in a sql query.

```
$query = 'insert into USERS (email, password, is_employee)    values ("' . $email . '", "' . $hash . '", 1)';
```

(db.php, line 617/618)

The $email variable is sanitized against sql injection.

```
$email = mysql_real_escape_string($email);
```
(db.php, line 605)

The $hash variable is calculated by php password_hash-function. This function calculates a hash value which couldn't be used for sql injection. It doesn't matter that $pass is a nonsecure user input.

```
$hash = password_hash($pass, PASSWORD_DEFAULT);
```
(db.php, line 616)

$email and $hash are sanitized or save inputs for the sql query so no vulnerability exist. It is a false positive.

SQL Injection - 2 - login_emp_db

The login_emp_db-function uses its input parameter $email in a sql query.

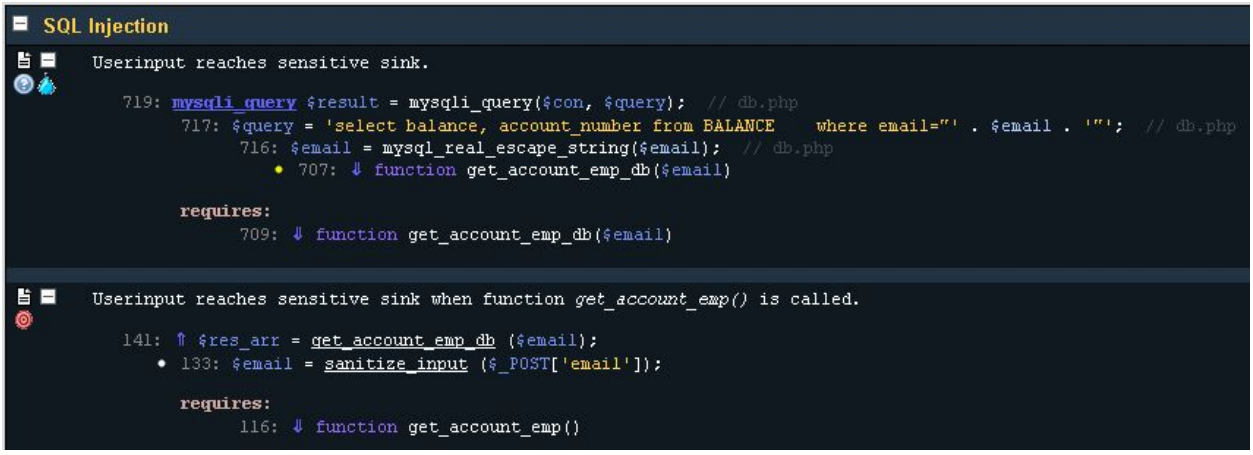
```
$query = 'select password, is_approved from USERS    where email="' . $email . '"    and is_employee=1';
```

(db.php, line 647-649)

$email is sanitized against sql injection so no vulnerability exist. It is a false positive.

```
$email = mysql_real_escape_string($email);
```
(db.php, line 646)

SQL Injection - 3 - get_account_emp_db



The login_emp_db-function uses its input parameter $email in a sql query.

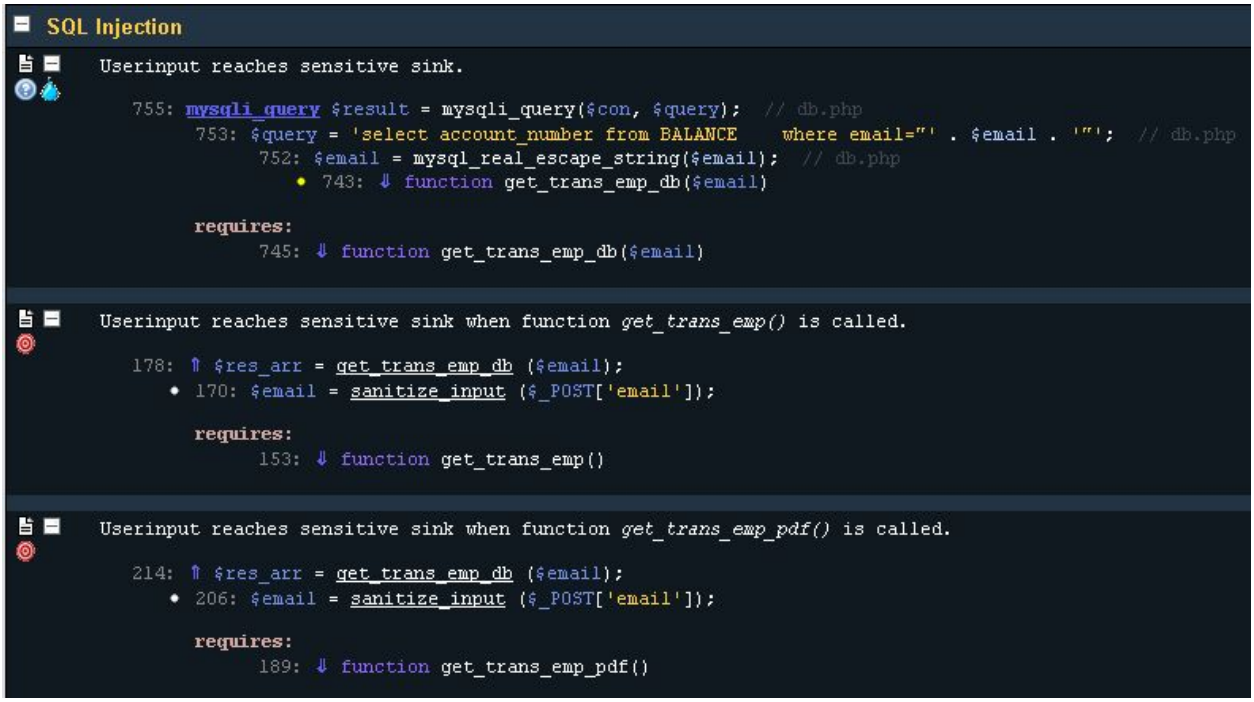
```
$query = 'select balance, account_number from BALANCE    where email="' . $email . '"';
```

(db.php, line 717/718)

$email is sanitized against sql injection so no vulnerability exist. It is a false positive.

```
$email = mysql_real_escape_string($email);
```
(db.php, line 716)

SQL Injection - 4 - get_trans_emp_db



The get_trans_emp_db-function uses its input parameter $email in a sql query.

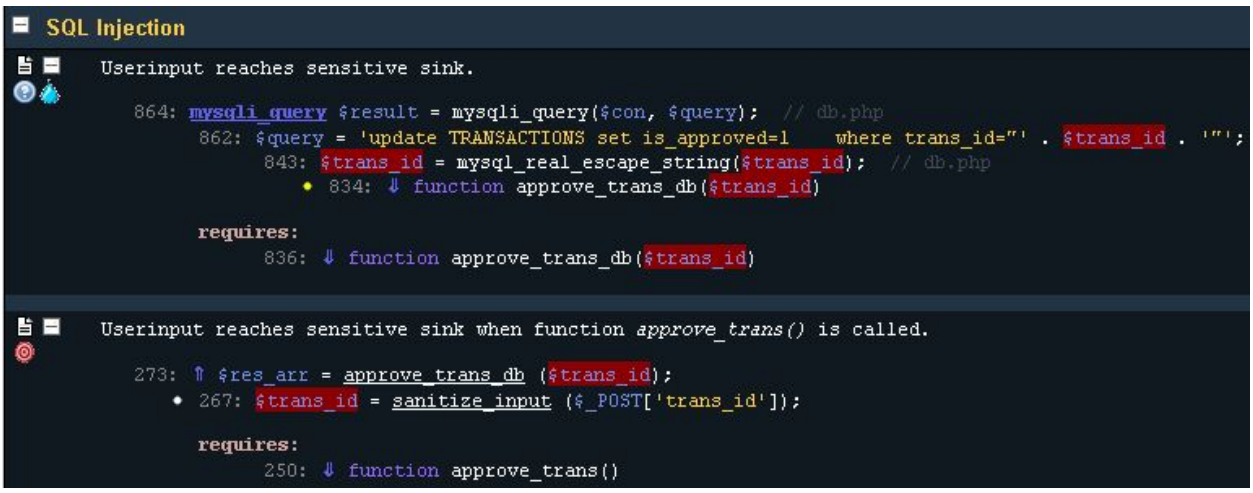
```
$query = 'select account_number from BALANCE    where email="' . $email . '"';
```
(db.php, line 753/754)
$email is sanitized against sql injection so no vulnerability exist. It is a false positive.
```
$email = mysql_real_escape_string($email);
```
(db.php, line 752)

SQL Injection - 5 - approve_trans_db

Item 11:

The approve_trans_db-function uses its input parameter $trans_id in a sql query.
```
$query = 'select account_num_src, account_num_dest, amount, is_approved from TRANSACTIONS    where trans_id="' . $trans_id . '"';
```
(db.php, line 844/845)
$trans_id is sanitized against sql injection so no vulnerability exist. It is a false positive.
```
$trans_id = mysql_real_escape_string($trans_id);
```
(db.php, line 843)

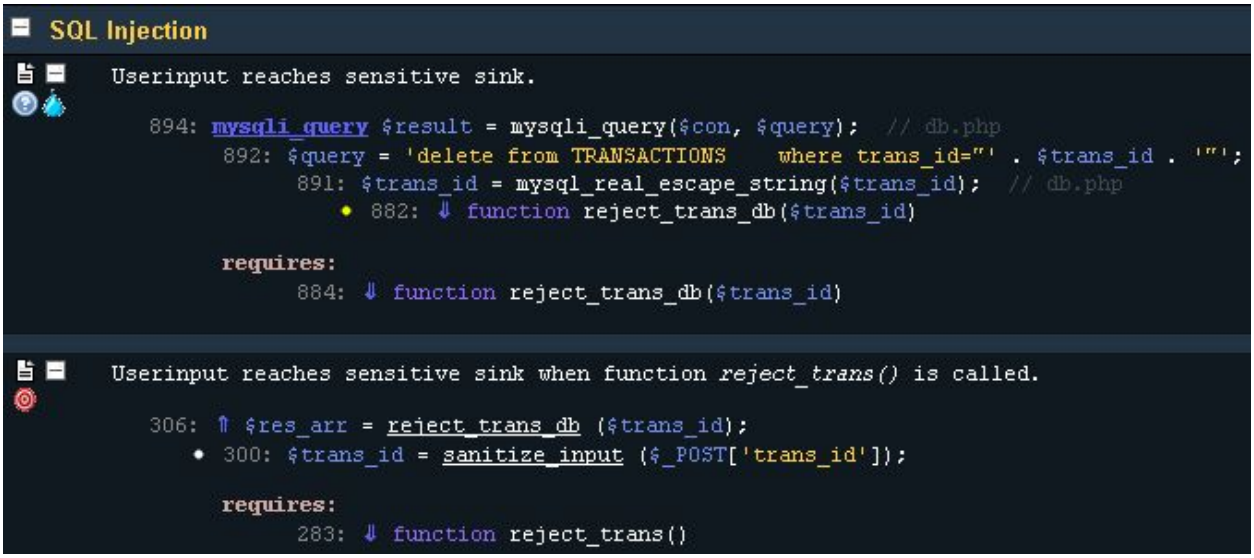
Item 12:



The approve_trans_db-function uses its input parameter $trans_id in a sql query.
```
$query = 'update TRANSACTIONS set is_approved=1    where trans_id="' . $trans_id . '"';
```
(db.php, line 862/863)
$trans_id is sanitized against sql injection so no vulnerability exist. It is a false positive.

```
$trans_id = mysql_real_escape_string($trans_id);
```
(db.php, line 843)

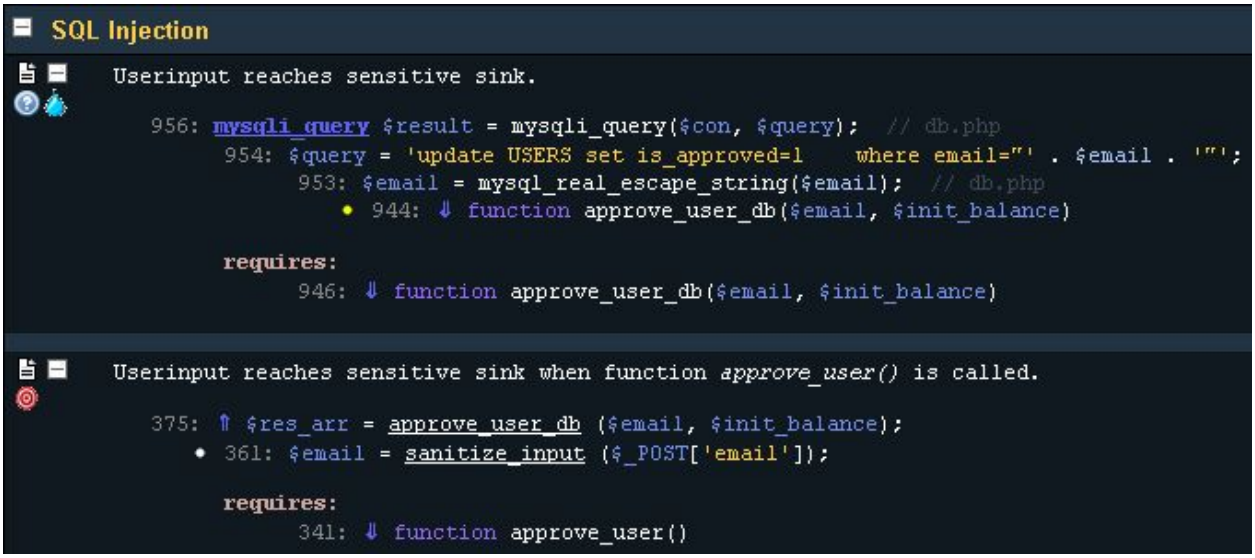
SQL Injection - 6 - reject_trans_db



The reject_trans_db-function uses its input parameter $trans_id in a sql query.
```
$query = 'delete from TRANSACTIONS    where trans_id="' . $trans_id . '"';
```
(db.php, line 892/893)
$trans_id is sanitized against sql injection so no vulnerability exist. It is a false positive.
```
$trans_id = mysql_real_escape_string($trans_id);
```
(db.php, line 891)

SQL Injection - 7 - approve_user_db

Item 14:



The approve_user_db-function uses its input parameter $email in a sql query.

```
$query = 'update USERS set is_approved=1    where email="' . $email . '"';
```

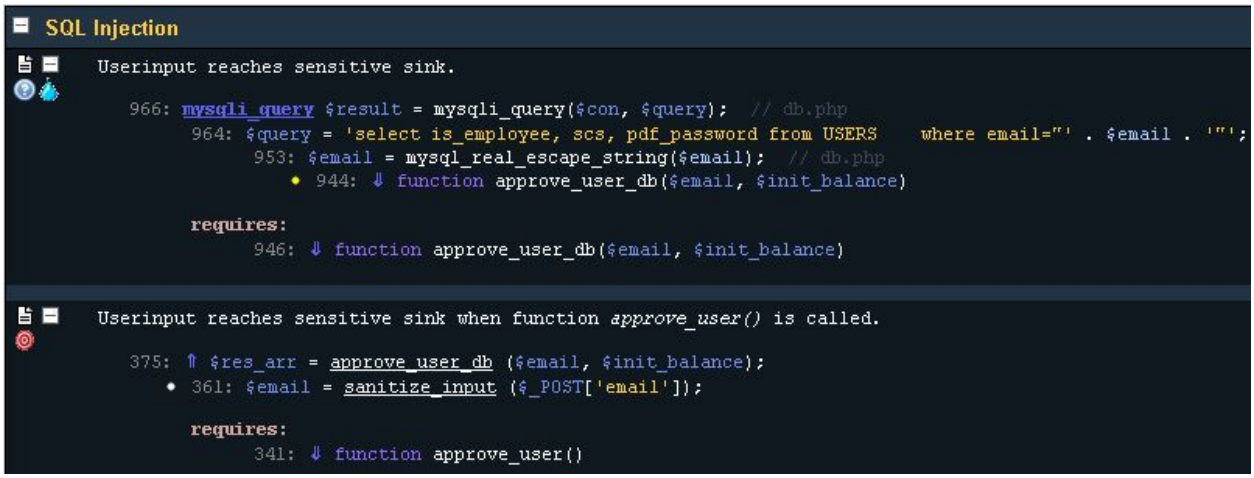
(db.php, line 954/955)

$email is sanitized against sql injection so no vulnerability exist. It is a false positive.

```
$email = mysql_real_escape_string($email);
```

(db.php, line 953)

Item 15:



The approve_user_db-function uses its input parameter $email in a sql query.

```
$query = 'select is_employee, scs, pdf_password from USERS    where email="' . $email . '"';
```

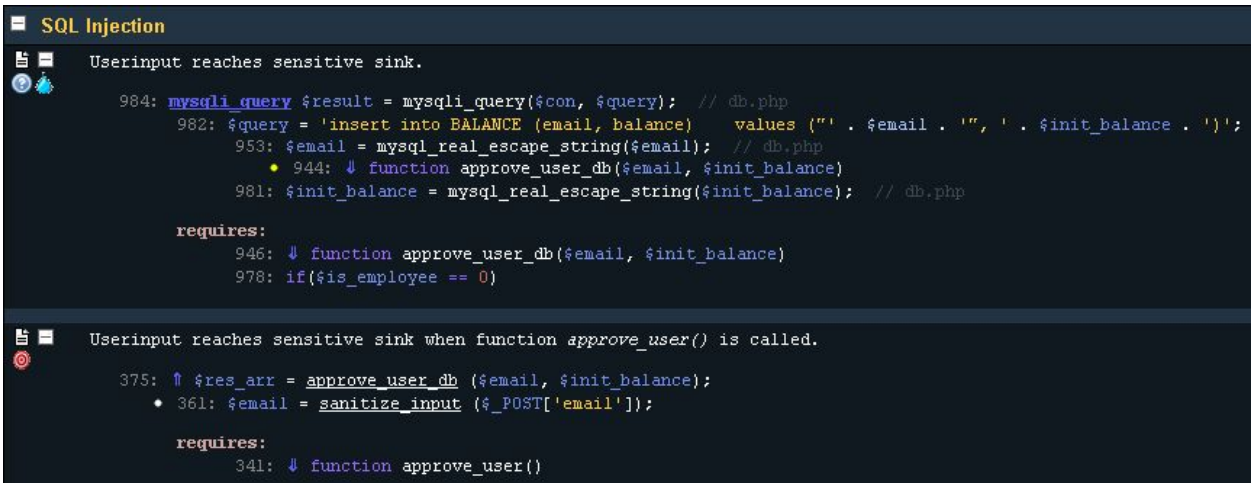
(db.php, line 964/965)

$email is sanitized against sql injection so no vulnerability exist. It is a false positive.

```
$email = mysql_real_escape_string($email);
```

(db.php, line 953)

Item 16:



The approve_user_db-function uses its input parameter $email and $init_balance in a sql query.

```
$query = 'insert into BALANCE (email, balance)    values ("' . $email . '", ' . $init_balance . ')';
```

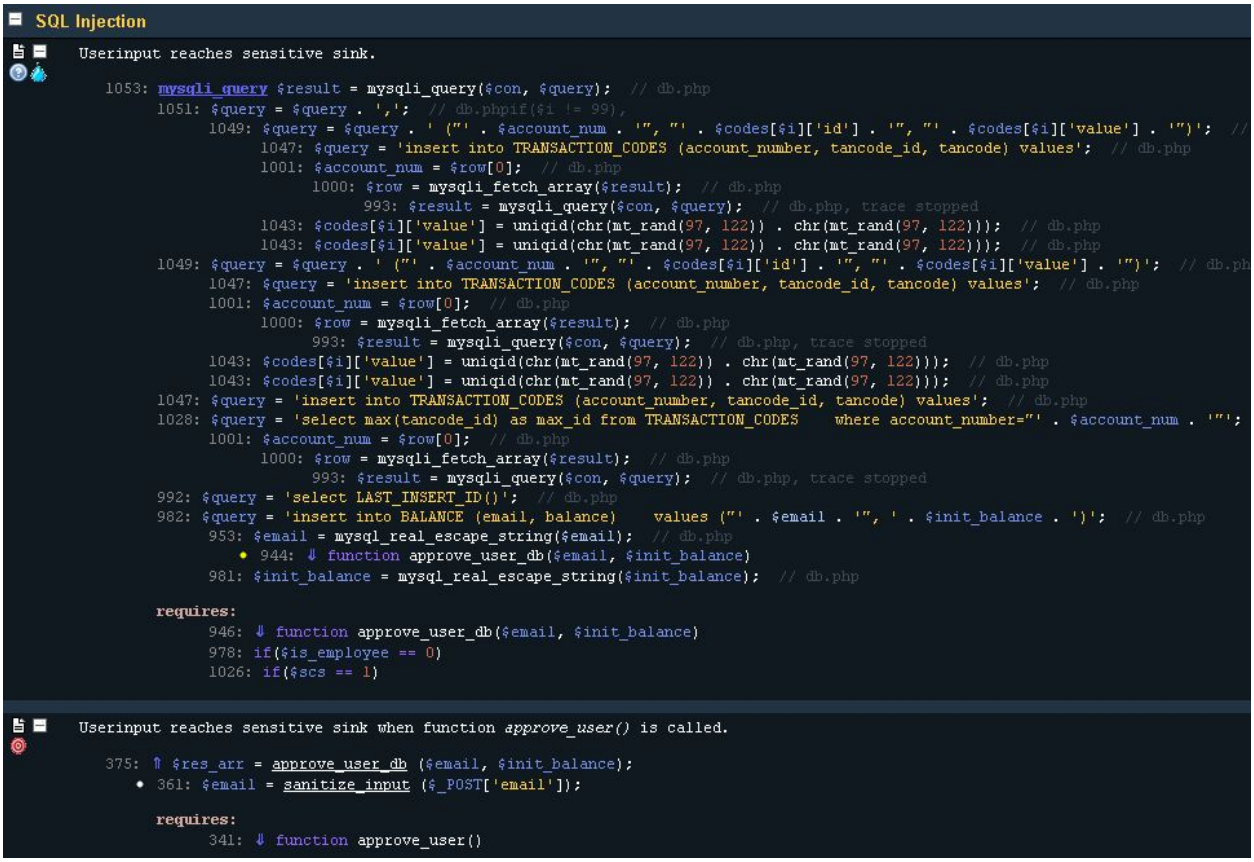
(db.php, line 982/983)

$email and $init_balance are sanitized against sql injection so no vulnerability exist. It is a false positive.

```
$email = mysql_real_escape_string($email);
```
(db.php, line 953)

```
$init_balance = mysql_real_escape_string($init_balance);
```
(db.php, line 981)

Item 17:



This entry reports multiple possible vulnerabilities.

query 1:

This query has been already discussed in “Item 16” (see above).

```
$query = 'insert into BALANCE (email, balance)    values ("' . $email . '", ' . $init_balance . ')';
```
(db.php, line 982)

query 2:

This query doesn’t have an user input so it isn’t vulnerable.

```
$query = 'select LAST_INSERT_ID()';
```
(db.php, line 992)

query 3:

This query is a concatenation of a constant and the variable $account_num.

```
$query = 'select max(tancode_id) as max_id from TRANSACTION_CODES    where account_number="' . $account_num . '"';
```

(db.php, line 1028)

The value of the $account_num variable is read from the database. The sql query to get this information is secure (see explanation of query 2) and therefore $account_num is also secure.

```
$query = 'select LAST_INSERT_ID()';
$result = mysqli_query($con, $query);
```

(db.php, line 992f)

```
$row = mysqli_fetch_array($result);
$account_num = $row[0];
```

(db.php, line 1000f)

query 4:

This query is a concatenation a constants, the secure $account_num variable (see explanation of query 2 and 3) and the $codes array. The security of this query depends on the security of the $codes array.

```
$query = 'insert into TRANSACTION_CODES (account_number, tancode_id, tancode) values';
for ($i = 0 ; $i < 100 ; $i++) {
    $query = $query . ' ("' . $account_num . '", "' . $codes[$i]['id'] . '", "' . $codes[$i]['value'] . '")';
    if ($i != 99)
        $query = $query . ',';
}
```

(db.php, line 1047ff)

The values of $codes['value'] are secure because they are generated by function that did not depend on user input. The values of $codes['id'] depend on the $start_id variable. The value of this variable is read from the database with "query 3". "query 3" is secure and therefore $start_id to.

```
$codes = array();
for ($i = 0 ; $i < 100 ; $i++) {
    $codes[$i]['id'] = $start_id + $i;
    $codes[$i]['value'] = uniqid(chr(mt_rand(97, 122)).chr(mt_rand(97, 122)));
}
```

(db.php, line 1039ff)

summery:

All four queries are not vulnerable. RIPS reports false positives.

Item 18:

The approve_user_db-function uses its input parameter $email and the calculated $scs_string and $scs_password in a sql query.

```
$query = 'update USERS set scs_string="' . $scs_string . '"
        , scs_password="' . $scs_password . '" where email="' . $email . '"';
```

(db.php, line 1071f)

$scs_password and $scs_string are calculated values without user input. So both values are secure and did not need to be sanitized.

```
$scs_password = rand(pow(10, 5), pow(10, 6) - 1);
```

(db.php, line 1065)

```
$scs_string = uniqid(chr(mt_rand(97, 122)) . chr(mt_rand(97, 122)));
```

(db.php, line 1068)

SQL Injection - 8 - reject_user_db

The reject_user_db-function uses its input parameter $email in a sql query.

```
$query = 'delete from USERS
        where email="' . $email . '"
        and is_employee=0
        and is_approved=0';
```

(db.php, line 1105ff)

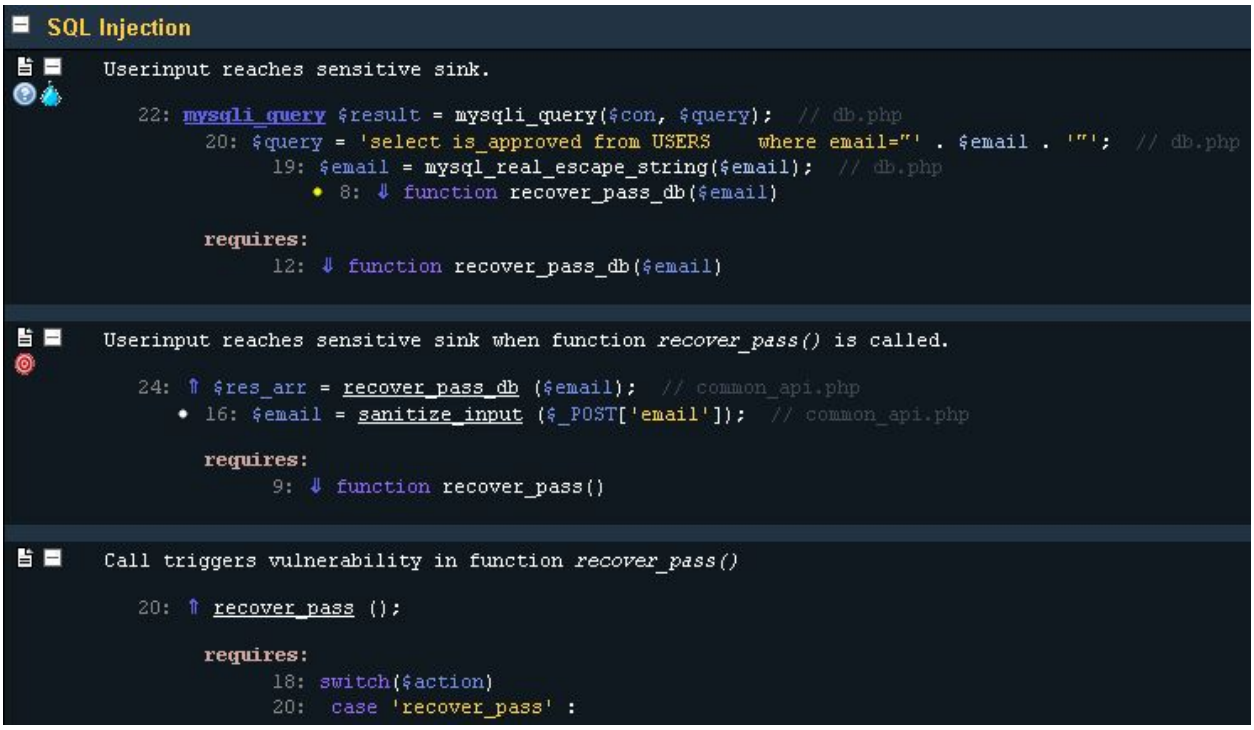
$email is sanitized against sql injection so no vulnerability exist. It is a false positive.

```
$email = mysql_real_escape_string($email);
```

(db.php, line 1104)

SQL Injection - 9 - recover_pass_db

Item 9:



The recover_pass_db-function uses its input parameter $email in a sql query.

```
$query = 'select is_approved from USERS    where email="' . $email . '"';
```
(db.php, line 20f)

The $email variable is sanitized against sql injection so no vulnerability exist. It is a false positive.

```
$email = mysql_real_escape_string($email);
```
(db.php, line 19)

Item 10:

The recover_pass_db-function uses its input parameter $email in a sql query. The query is a concatenated string of constants, the local $token variable, the global $PASSREC_TOKEN_DURATION variable and the input parameter $email. The global $PASSREC_TOKEN_DURATION is configured from an administrator so it should be secure.

```
$query = 'update USERS set password_token= "' . $token . '"
        , exp_date=ADDTIME(now(), "' . $PASSREC_TOKEN_DURATION . '"), was_used=0
        where email="' . $email . '"';
```

(db.php, line 36f)

The local $token variable is calculated without user input so it is secure.

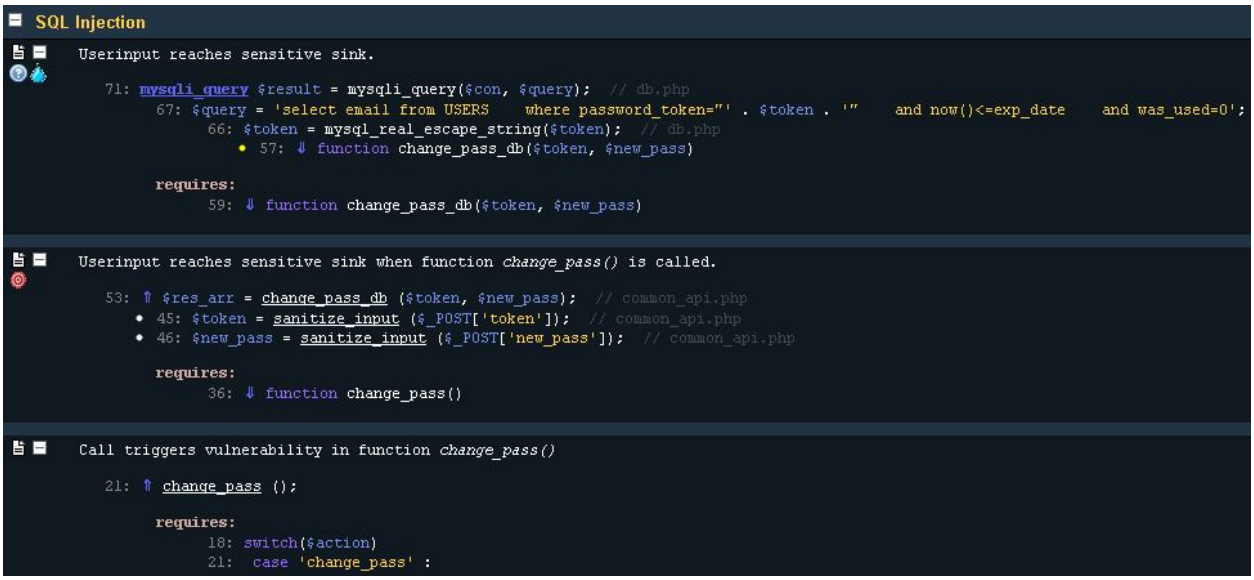
```
$token = sha1(openssl_random_pseudo_bytes(20));
```
(db.php, line 35)

The $email variable is sanitized against sql injection so no vulnerability exist. It is a false positive.

```
$email = mysql_real_escape_string($email);
```
(db.php, line 19)

SQL Injection - 10 - change_pass_db

Item 11:



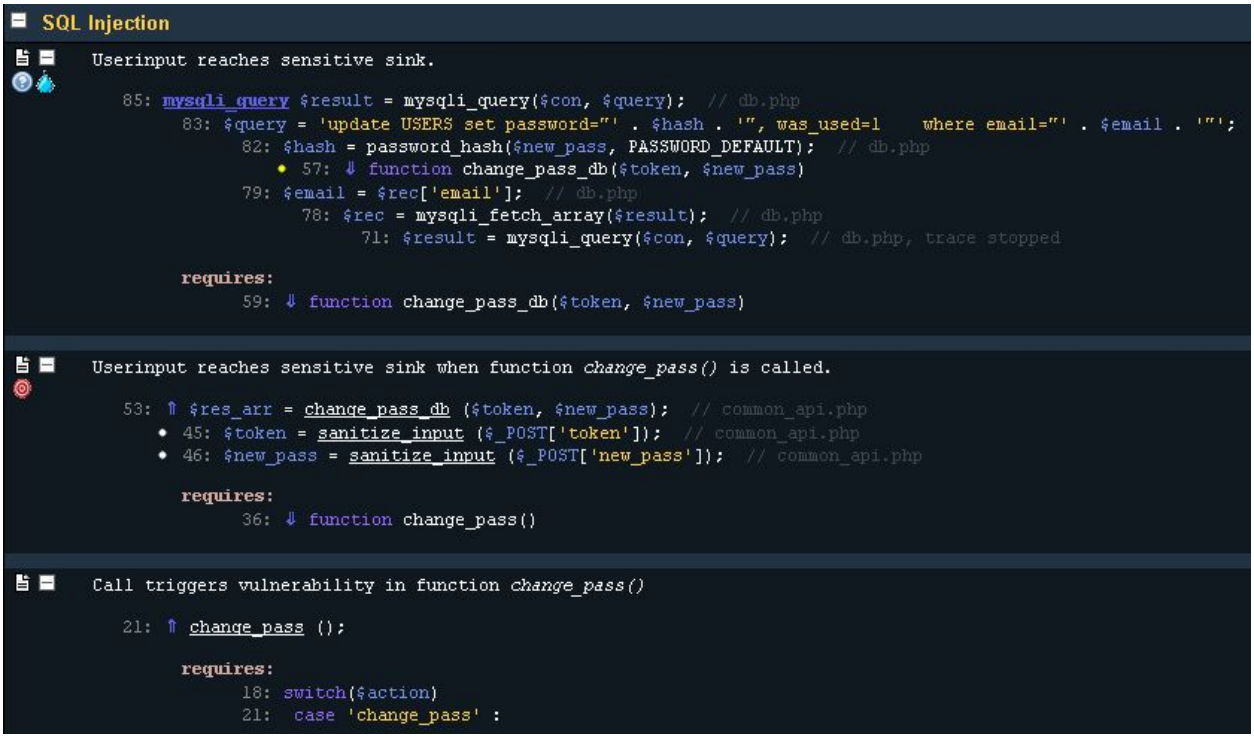
The change_pass_db-function uses its input parameter $token in a sql query.

```
$query = 'select email from USERS
        where password_token="' . $token . '"
        and now()<=exp_date
        and was_used=0';
```
(db.php, line 67ff)

The $token variable is sanitized against sql injection so no vulnerability exist. It is a false positive.

```
$token = mysql_real_escape_string($token);
```
(db.php, line 66)

Item 12:



The change_pass_db-function uses its input parameter $email and the $hash variable in a sql query.

```
$query = 'update USERS set password="' . $hash . '", was_used=1    where email="' . $email . '"';
```

(db.php, line 83f)

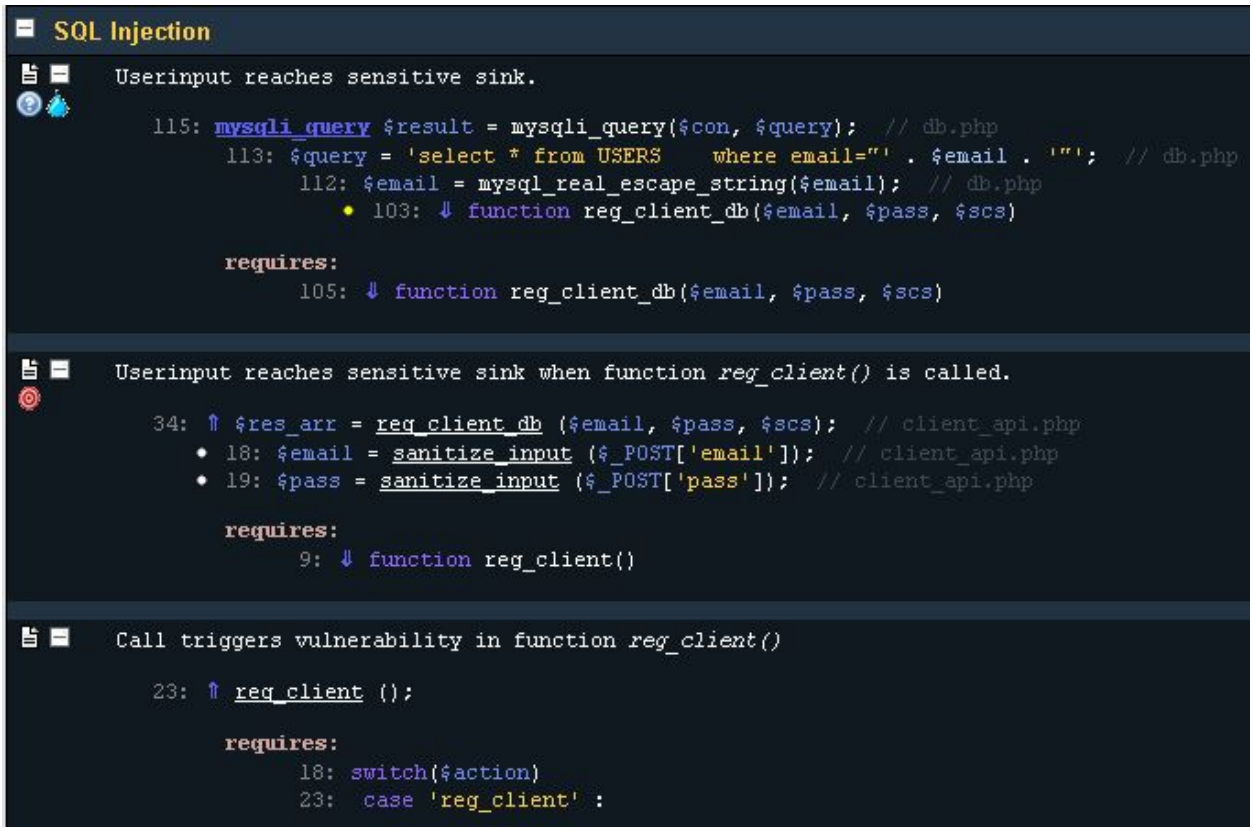
The $hash variable is calculated by php password_hash-function. This function calculates a hash value which couldn't be used for sql injection. It doesn't matter that $new_pass is a nonsecure user input.

```
$hash = password_hash($new_pass, PASSWORD_DEFAULT);
```
(db.php, line 82)

The $email variable is read from the database. The sql statement which has been used is secure (see explanation above for item 11) so $email is also secure. $email and $hash are secure inputs for the sql query so no vulnerability exist. It is a false positive.

SQL Injection - 11 - req_client_db

Item 13:



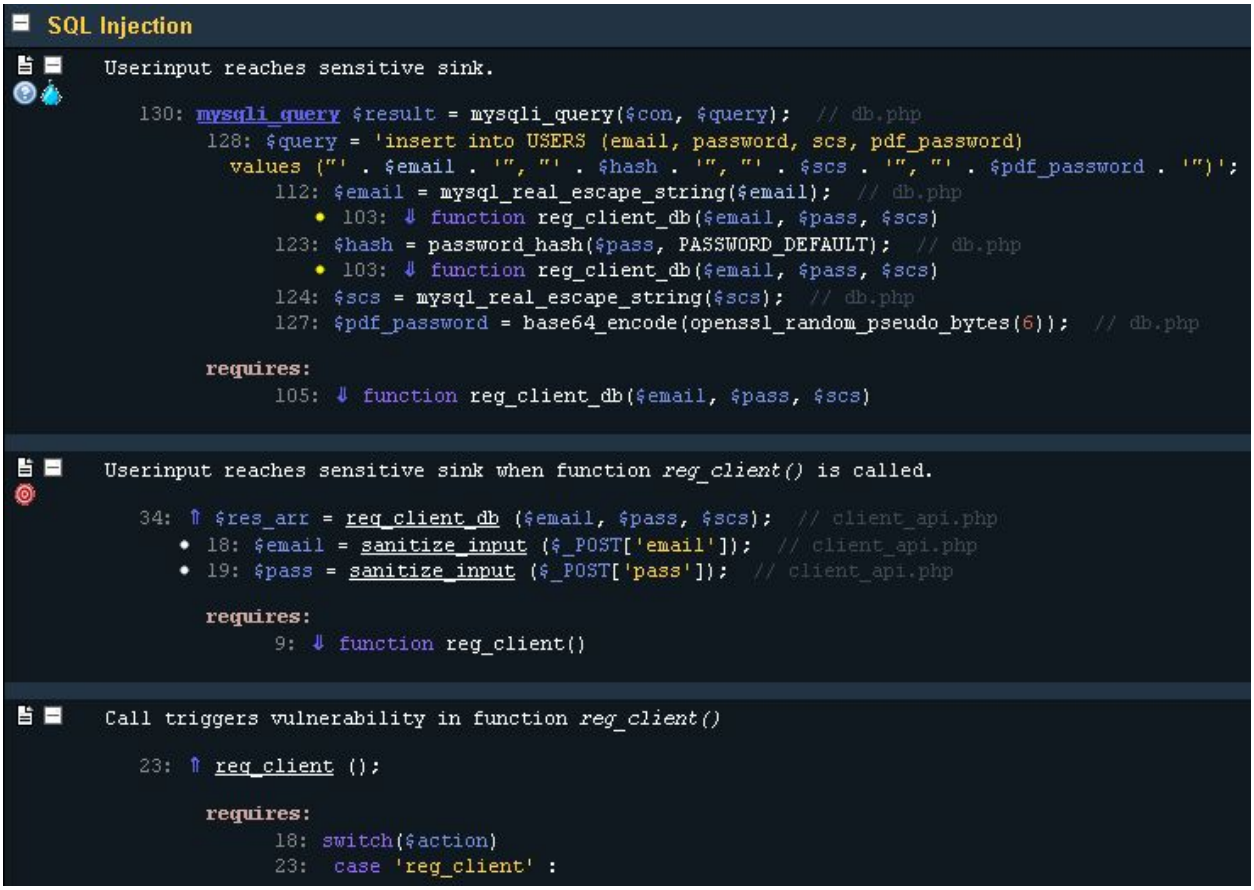
The req_client_db-function uses its input parameter $email in a sql query.

```
$query = 'select * from USERS    where email="' . $email . '"';
```
(db.php, line 113)

The $email variable is sanitized against sql injection so no vulnerability exist. It is a false positive.

```
$email = mysql_real_escape_string($email);
```
(db.php, line 112)

Item 14:



The req_client_db-function uses its input parameter $email and $scs and the $hash and $pdf_password variable in a sql query.

```
$query = 'insert into USERS (email, password, scs, pdf_password)
        values ("' . $email . '", "' . $hash . '", "' . $scs . '", "' . $pdf_password . '")';
```

The $hash variable is calculated by php password_hash-function. This function calculates a hash value which couldn't be used for sql injection. It doesn't matter that $pass is a nonsecure user input.

```
$hash = password_hash($pass, PASSWORD_DEFAULT);
```
(db.php, line 123)

The $pdf_password variable is calculated without user input so it is secure.

```
$pdf_password = base64_encode(openssl_random_pseudo_bytes(6));
```
(db.php, line 127)

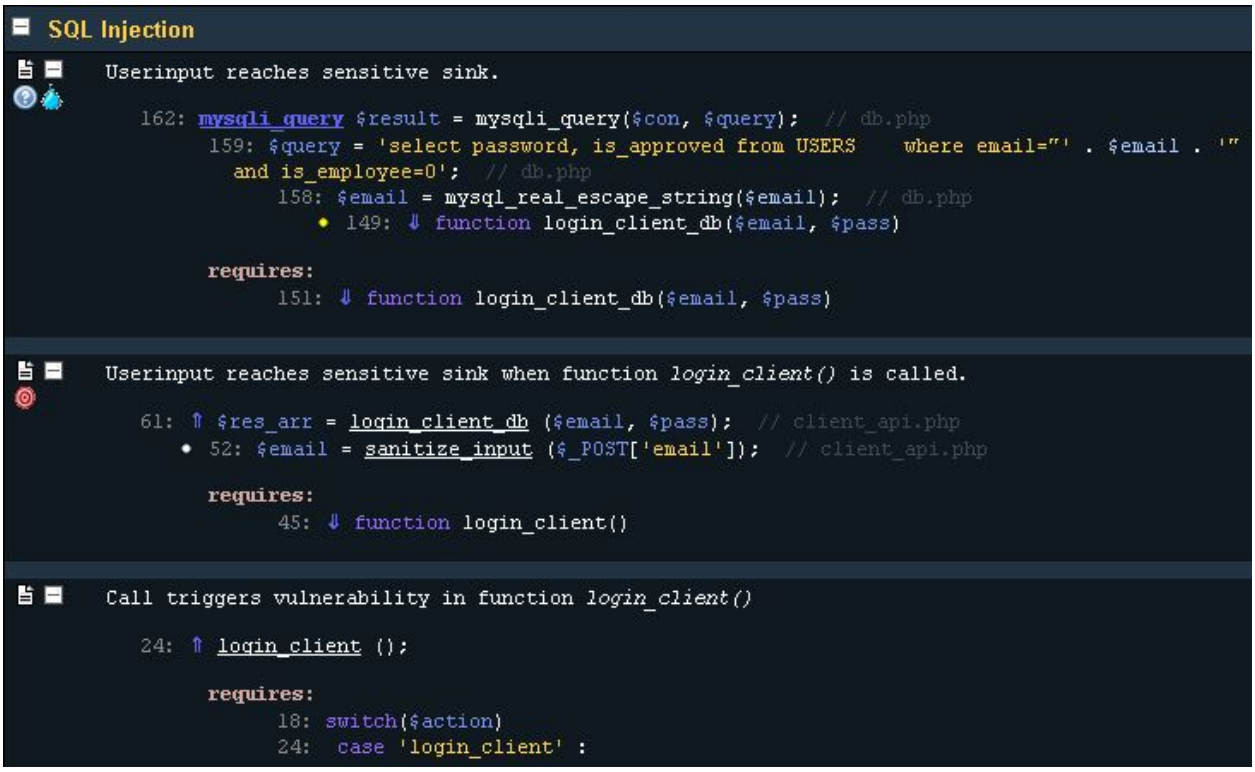
The input parameter $email and $scs are sanitized against sql injection so no vulnerability exist. It is a false positive.

```
$email = mysql_real_escape_string($email);
```
(db.php, line 112)

```
$scs = mysql_real_escape_string($scs);
```
(db.php, line 124)

SQL Injection - 12 - login_client_db

Item 15:



The login_client_db-function uses its input parameter $email n a sql query.

```
$query = 'select password, is_approved from USERS
        where email="' . $email . '"
        and is_employee=0';
```

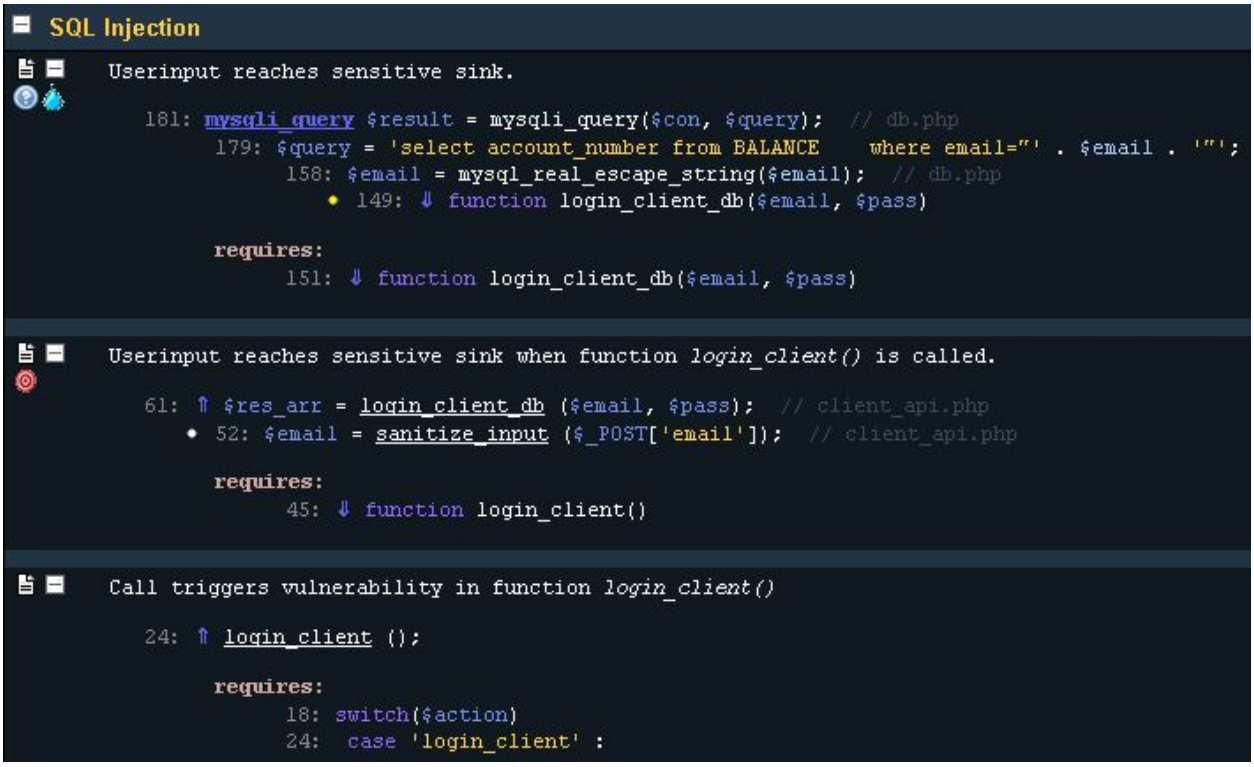
(db.php, line 159ff)

The input parameter $email is sanitized against sql injection so no vulnerability exist. It is a false positive.

```
$email = mysql_real_escape_string($email);
```

(db.php, line 158)

Item 16:



The login_client_db-function uses its input parameter $email n a sql query.

`$query = 'select account_number from BALANCE where email="' . $email . '"';` (db.php, line 179)

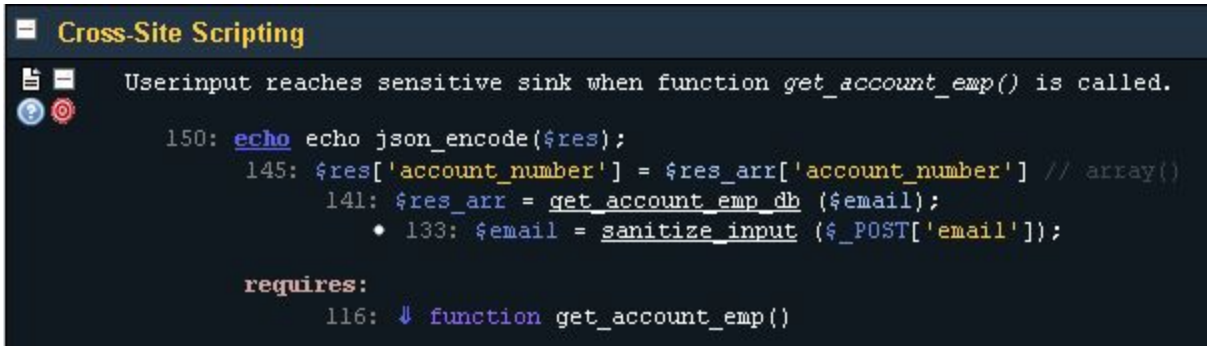
The input parameter $email is sanitized against sql injection so no vulnerability exist. It is a false positive.

`$email = mysql_real_escape_string($email);` (db.php, line 158)

Cross-Site Scripting - 1 - print_debug_message

See explanation of Cross-Site-Scripting of RUN 1.

Cross-Site Scripting - 2 - get_account_emp



The get_account_emp-function returns a JSON encoded array. This JSON-object may contain unsecure user input.

```
echo json_encode($res);
```
(employee_api.php, line 150)

The $res variable is an array, which contains constants and the values $res_arr['balance'] and $res_arr['account_number'].

```
$res = array('status' => 'true',
        'message' => null,
        'balance' => $res_arr['balance'],
        'account_number' => $res_arr['account_number']);
```
(employee_api, line 145ff)

The $res_arr-array is filled by the get_account_emp_db-function.

```
$res_arr = get_account_emp_db($email);
```
(employee_api, line 141)

```
return array('status' => true,
        'balance' => $balance,
        'account_number' => $account_number);
```
(db.php, line 738ff)

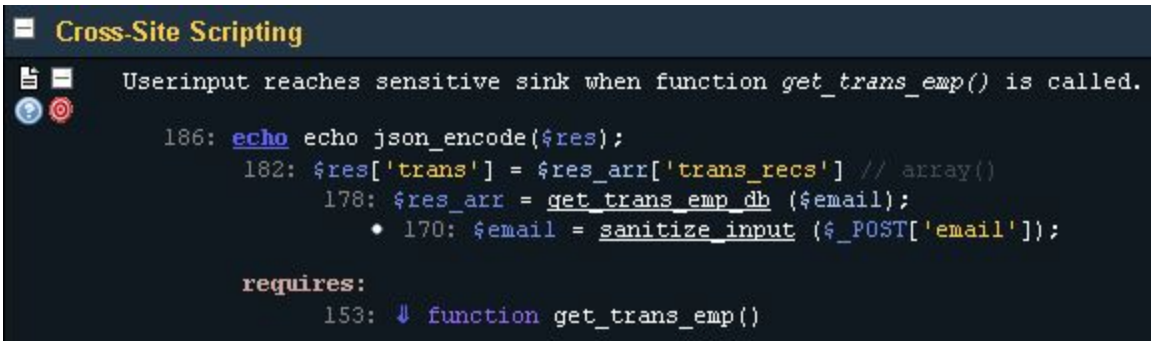
The get_account_emp_db-function reads the balance and account number from the database. The sql statement is a concatenated string of constants and the input parameter $email. This parameter is sanitized against sql injection. Therefore the values of $balance and $account_number are trustworthy.

```
$email = mysql_real_escape_string($email);
$query = 'select balance, account_number from BALANCE
      where email="' . $email . '"';
$result = mysqli_query($con, $query);
```
(db.php, line 716ff)

```
$row = mysqli_fetch_array($result);
$balance = $row['balance'];
$account_number = $row['account_number'];
```
(db.php, line 726ff)

There is no vulnerability. It is a false positive.

Cross-Site Scripting - 3 - get_trans_emp



The get_account_emp-function returns a JSON encoded array. This JSON-object may contain unsecure user input.

```
echo json_encode($res);
```
(employee_api.php, line 186)

The $res variable is an array, which contains constants and the value $res_arr[‘trans_recs’].

```
$res = array('status' => 'true',
        'message' => null,
        'trans' => $res_arr['trans_recs']);
```
(employee_api.php, line 182ff)

The $res_arr-array is filled by the get_trans_emp_db-function.

```
$res_arr = get_trans_emp_db($email);
```
(employee_api.php, line 178)

```
return array('status' => true,
        'account_num' => $account_num,
        'trans_recs' => $trans_recs);
```
(db.php, line 791ff)

The get_trans_emp_db-function reads the account number and the transaction records from the database. The account number and transaction records are read in two separate sql statements.

Firstly the account number is determine. The output of the sql statement is secure because the input parameter $email is escaped.

```
$email = mysql_real_escape_string($email);
$query = 'select account_number from BALANCE
      where email="' . $email . '"';
$result = mysqli_query($con, $query);
```
(db.php, line752ff)

```
$row = mysqli_fetch_array($result);
$account_num = $row['account_number'];
```
(db.php, line 762f)

Secondly the transaction records are read from the database. The output of the sql statement is secure because the input parameter $account_num has been read from the database before and is nevertheless escaped.

```
$account_num = mysql_real_escape_string($account_num);
$query = 'select trans_id, account_num_src, account_num_dest, amount, description,
            date, is_approved from TRANSACTIONS
      where account_num_src="' . $account_num . '"
      order by trans_id';
$result = mysqli_query($con, $query);
```

(db.php, line 766)

The sql statement was secure but its output data was not sanitized. If the description contains a script for cross-site scripting, it is vulnerability. All the other data couldn’t be used for attacks because they are of type integer, timestamp or boolean.

```
$trans_recs = array();
while ($rec = mysqli_fetch_array($result)) {
    $trans_rec = array($rec['trans_id'],
                $rec['account_num_dest'],
                $rec['amount'],
                $rec['description'],
                $rec['date'],
                $rec['is_approved']);
    array_push($trans_recs, $trans_rec);
}
```

(db.php, line 773)

The only way to create a transaction tupel in the database is by using the transfer_money-function. The $description variable has been sanitized again sql injection but maybe not against cross-site scripting. So we need to have a closer look at this.

```
$description = mysql_real_escape_string($description);
$query = 'insert into TRANSACTIONS (account_num_src, account_num_dest, amount, description, is_approved)
          values ("' . $account_num_src . '", "'
                 . $account_num_dest . '", "'
                 . $amount . '", "'
                 . $description . '", "'
                 . $is_approved . '")';
```

The transfer_money-function is called by set_trans_form_db-, set_trans_file_db- and approve_trans_db-function.

approve_trans_db:

The approve_trans_db-function calls the transfer_money-function with an empty description. Therefore no vulnerability could be created.

```
$res_arr = transfer_money($row['account_num_src'], $row['account_num_dest'], $row['amount'], '', 1);
```

(db.php, line 858)

set_trans_file_db:

The set_trans_file_db-function calls the transfer_money-function with the third column of the $params-array. The $params-array is an unsanitized input parameter of the function.

```
for ($i = 0 ; $i < count($params)-1 ; $i++) {
    $res_arr = transfer_money($account_num_src, $params[$i][0], $params[$i][1], $params[$i][2], 0);
```

(db.php, line 473ff)

The set_trans_file_db-function is only called by the set_trans_file-function.

```
$res_arr = set_trans_file_db($email, $account_num_src, $tancode_id, $tancode_value, $params, 0);
```

(client_api.php, line 348)

```
$res_arr = set_trans_file_db($email, $account_num_src, $tancode_id, $scs_token, $params, $file_contents);
```

(client_api.php, line 360)

The $params variable is set by calling the parse_file-function.

```
$params = parse_file($res_arr['filename']);
```

(client_api.php, line 330)

This function reads the file which is uploaded by the user into the array. Each line of the file is split into its words by using the str_word_count-function.

```
while ($s = fgets($handle)) {
    $words = str_word_count($s, 1, '1234567890');
    array_push($params, $words);
}
```

(aux_func.php, line 275)

This function also removes every special character like <,>,/ and \. Therefore it isn't possible to insert a cross-site script and no vulnerability could be created.

Example:

code	print_r(str_word_count('123 123 <script>alert("hallo")</script>', 1, '1234567890'))
output	Array ([0] => 123 [1] => 123 [2] => script [3] => alert [4] => hallo [5] => script)

set_trans_form_db:

The set_trans_from_db-function calls the transfer_money-function with an unsanitized description.

```
$res_arr = transfer_money($account_num_src, $account_num_dest, $amount, $description, 0);
```

(db.php, line 396)

The set_trans_from_db-function is only called by the set_trans_from-function.

```
$res_arr = set_trans_form_db($email, $account_num_src, $account_num_dest, $amount, $tancode_id, $tancode_value, $description);
```

(client_api.php, line 294)

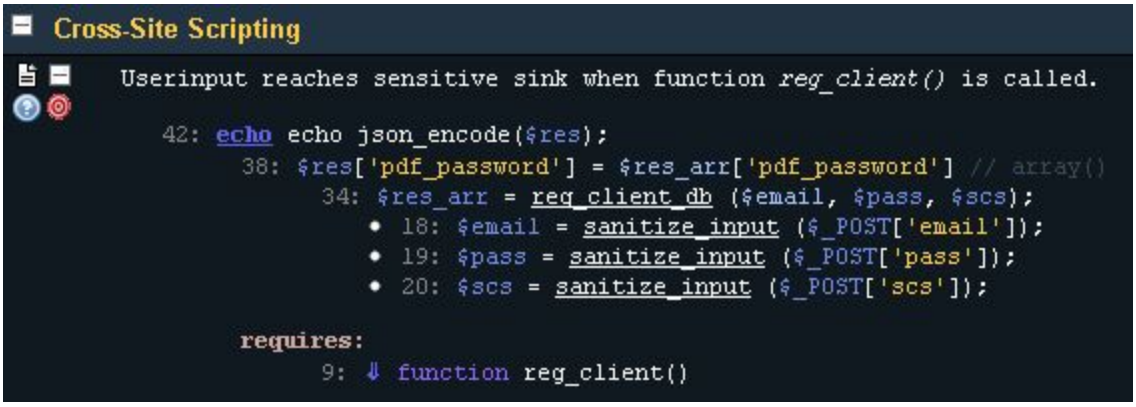
This function sanitizes the description so no vulnerability could be created.

```
$description = sanitize_input($_POST['description']);
```

(client_api.php, line 271)

As a conclusion: there is no vulnerability. It is a false positive.

Cross-Site Scripting - 4 - req_client



The req_client-function returns a JSON encoded array. This JSON-object may contain unsecure user input.

```
echo json_encode($res);
```
(client_api.php, line 42)

The $res variable is an array, which contains constants and the value $res_arr['pdf_password'].

```
$res = array('status' => 'true',
             'message' => null,
             'pdf_password' => $res_arr['pdf_password']);
```
(client_api.php, line 38ff)

The $res_arr-array is filled by the get_client_db-function.

```
$res_arr = reg_client_db($email, $pass, $scs);
```
(client_api.php, line 34)

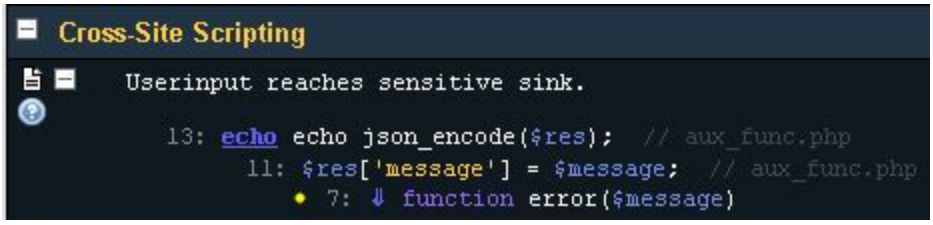
The req_client_db-function contains multiple sql query. These query are secure (see above SQL Injection 11 - req_client_db).

The $res_arr['pdf_password'] variable is calculated without user input, so it is secure.

```
$pdf_password = base64_encode(openssl_random_pseudo_bytes(6));
```
(db.php, line 123)

Therefore no vulnerability exist. It is a false positive.

Cross-Site Scripting - 5 - error



The error-function returns a JSON encoded array. This JSON-object may contain unsecure user input.

```
echo json_encode($res);
```
(aux_func.php, line 13)

The $res variable is an array, which contains constants and the input parameter $message.

```
$res['status'] = 'false';
$res['message'] = $message;
```
(aux_func.php, line 10f)

The input parameter $message isn't sanitized by the error-function. This could lead to a vulnerability. The callee of the error-function needs to make sure that the $message is secure. So we need to have a closer look at the function stack trace. In the whole application the error-function is called 232 times by other functions. Many of them are secure because they use only constants. The other ones are reported by RIPS and need to be checked.

There are 17 reported vulnerabilities, which are all structured equally. So we will only discuss two examples in detail.

recover_pass:

```
Userinput reaches sensitive sink when function recover_pass() is called.

    26: ⇑ error ($res_arr['err_message']);  // common_api.php
    24: $res_arr = recover_pass_db ($email);  // common_api.php
    • 16: $email = sanitize_input ($_POST['email']);  // common_api.php

    requires:
        25: if($res_arr['status'] == false)
        9: ⇓ function recover_pass()
```

The recover_pass-function uses the error message of the recover_pass_db-function without sanitizing it. So we need to have a closer look to all possible outputs of this function.

```
$res_arr = recover_pass_db($email);
if ($res_arr['status'] == false)
    return error($res_arr['err_message']);
```

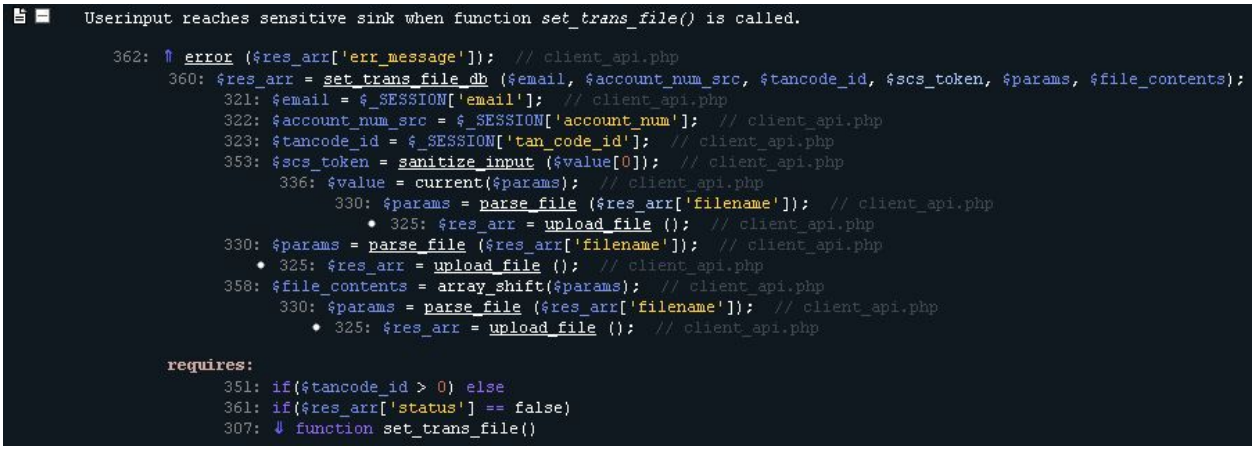
(common_api.php, line 24ff)

The function only uses constants in the output where ‘status’ = false. Therefor no vulnerability exists.

set_trans_file:

```
Userinput reaches sensitive sink when function set_trans_file() is called.

    327: ⇑ error ($res_arr['err_message']);  // client_api.php
    • 325: $res_arr = upload_file ();  // client_api.php

    requires:
        326: if($res_arr['status'] == false)
        307: ⇓ function set_trans_file()
```

```
Userinput reaches sensitive sink when function set_trans_file() is called.

    350: ⇑ error ($res_arr['err_message']);  // client_api.php
    348: $res_arr = set_trans_file_db ($email, $account_num_src, $tancode_id, $tancode_value, $params, 0);
        321: $email = $_SESSION['email'];  // client_api.php
        322: $account_num_src = $_SESSION['account_num'];  // client_api.php
        323: $tancode_id = $_SESSION['tan_code_id'];  // client_api.php
        344: $tancode_value = sanitize_input ($value[0]);  // client_api.php
            336: $value = current($params);  // client_api.php
                330: $params = parse_file ($res_arr['filename']);  // client_api.php
                • 325: $res_arr = upload_file ();  // client_api.php
        330: $params = parse_file ($res_arr['filename']);  // client_api.php
        • 325: $res_arr = upload_file ();  // client_api.php

    requires:
        342: if($tancode_id > 0)
        349: if($res_arr['status'] == false)
        307: ⇓ function set_trans_file()
```



The set_trans_file-function uses the error message of the upload_file- and two times set_trans_file_db-function without sanitizing it. So we need to have a closer look to all possible outputs of these functions.

```
$res_arr = upload_file();
if ($res_arr['status'] == false)
    return error($res_arr['err_message']);
```

(common_api.php, line 325ff)

```
$res_arr = set_trans_file_db($email, $account_num_src, $tancode_id, $tancode_value, $para
if ($res_arr['status'] == false)
    return error($res_arr['err_message']);
```

(common_api.php, line 348ff)

```
$res_arr = set_trans_file_db($email, $account_num_src, $tancode_id, $scs_token, $params, $
if ($res_arr['status'] == false)
    return error($res_arr['err_message']);
```

(common_api.php, line 360ff)

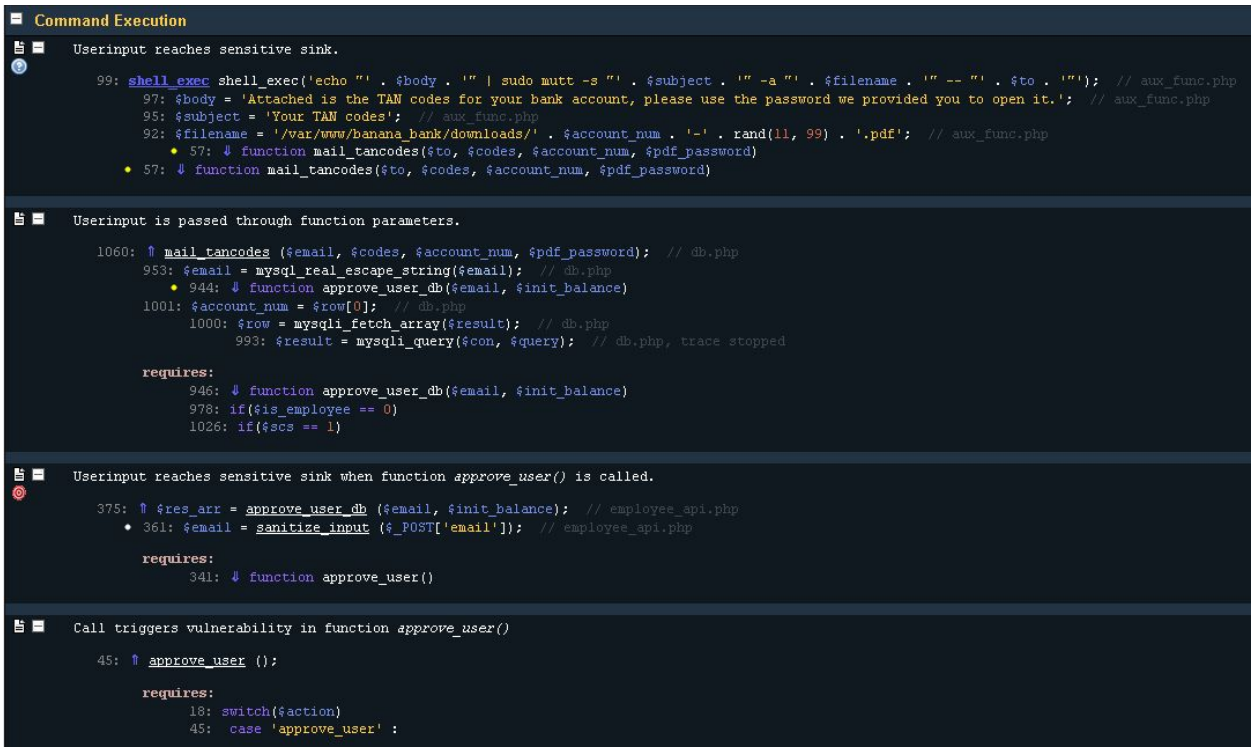
The functions only use constants in the output where ‘status’ = false. Therefor no vulnerability exists.

summary:

source: serve_req = serve_request.php
empl = employee_api.php

source	function	used subfunction	vulnerable?
serve_req	recover_pass	recover_pass_db	no
serve_req	change_pass	change_pass_db	no
serve_req	req_client	req_client_db	no
serve_req	login_client	login_client_db	no
serve_req	set_trans_from	set_trans_form_db	no
serve_req	set_trans_file	upload_file set_trans_file_db	no no
serve_req, empl	req_emp	req_emp_db	no
serve_req, empl	login_emp	login_emp_db	no
serve_req, empl	get_account_emp	get_account_emp_db	no
serve_req, empl	get_trans_emp	get_trans_emp_db	no
serve_req, empl	get_trans_emp_pdf	get_trans_emp_db	no
serve_req, empl	approve_trans	approve_trans_db	no
serve_req, empl	reject_trans	reject_trans_db	no
serve_req, empl	approve_user	approve_user_db	no
serve_req, empl	reject_user	reject_user_db	no

Command Execution - 1 - mail_tancodes



The function mail_tancodes executes a shell command which is a concatenation of the $body, $subject, $filename and $to.

```
shell_exec('echo "' . $body . '" | sudo mutt -s "' . $subject . '" -a "' . $filename . '" -- "' . $to . '"');
```

$body and $subject are constants, so there is no need of sanitizing.

```
$body = 'Attached is the TAN codes for your bank account, please use the password we provided you to open it.';
$subject = 'Your TAN codes';  // aux_func.php
```

$filename is a concatenated string of constants, a save function and $account_num. The $filename is insecure if $account_num isn't sanitized.

```
$filename = '/var/www/banana_bank/downloads/' . $account_num . '-' . rand(11, 99) . '.pdf';
```

$account_num and $to are unsanitized input parameter of the function. So we need to have a closer look at the function stack trace.

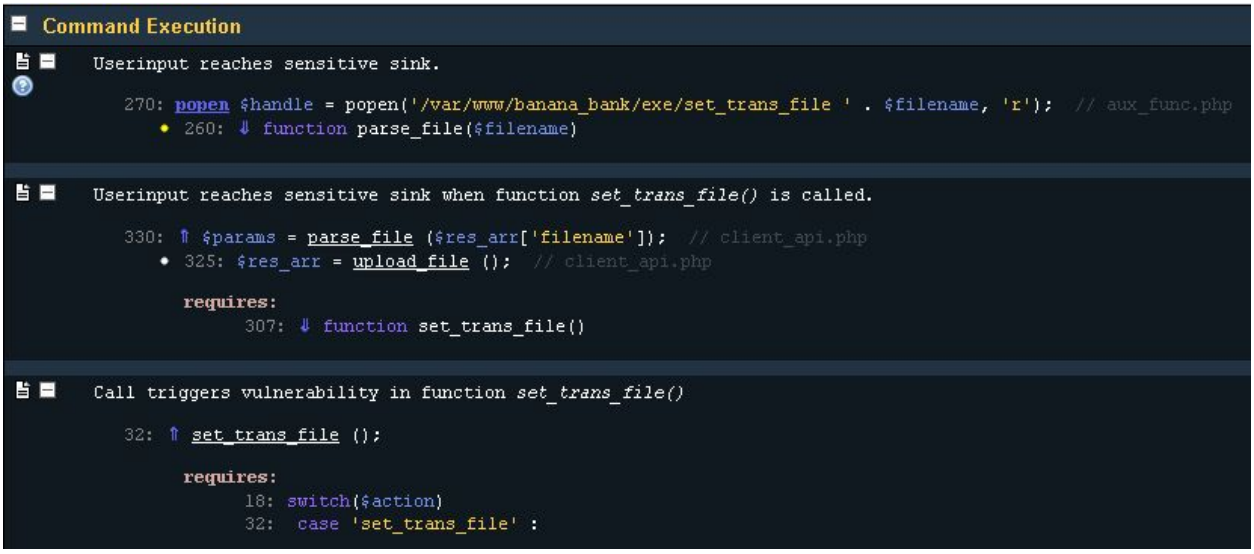
Function Stack Trace:

user request
⇒ approve_user
→ $email will be the input parameter $to of mail_tancodes
⇒ sanitize_input($email)
→ offers only little protection against command execution

⇒ **filter_var($email, FILTER_VALIDATE_EMAIL)**
→ format checking; email format couldn't be used for command execution
⇒ approve_user_db
⇒ mysql_real_escape_string($email)
→ sanitizing against sql injection; no extra protection
→ **$account_num is read from database. Database field is an integer. Therefore no sanitizing is necessary**
⇒ mail_tancodes

$account_num is read from database and the format of $email is checked, so probably no vulnerability exists. If you manage to construct a valid email address which is a useful shell command then the application is vulnerable.

Command Execution - 2 - parse_file



The parse_file-function opens a process and executes a shell command. The command is a concatenated string of a constant and the $filename variable. The $filename is an unsanitized input parameter, so we need to have a closer look.

```
popen $handle = popen('/var/www/banana_bank/exe/set_trans_file ' . $filename, 'r');
```

The parse_file-function is only called by the set_trans_file-function.

```
$params = parse_file ($res_arr['filename']);
```

The value of $res_arr['filename'] is set by the upload_file-function.

```
$res_arr = upload_file ();
```
(
```
return array('status' => true, 'filename' => $target);
```
)

The $target variable is a concatenation of a constant and the $name variable. The $name variable is sanitized by the sanitize_input-function and the php $_FILES functionality.

```
$name = sanitize_input($_FILES['uploadFile']['name']);
$target = '/var/www/banana_bank/uploads/' . $name;
```
(aux_func.php, line 246f)

If the $target variable isn't a valid filename, the $move_uploaded_file would fail and the upload_file-function would report an error.

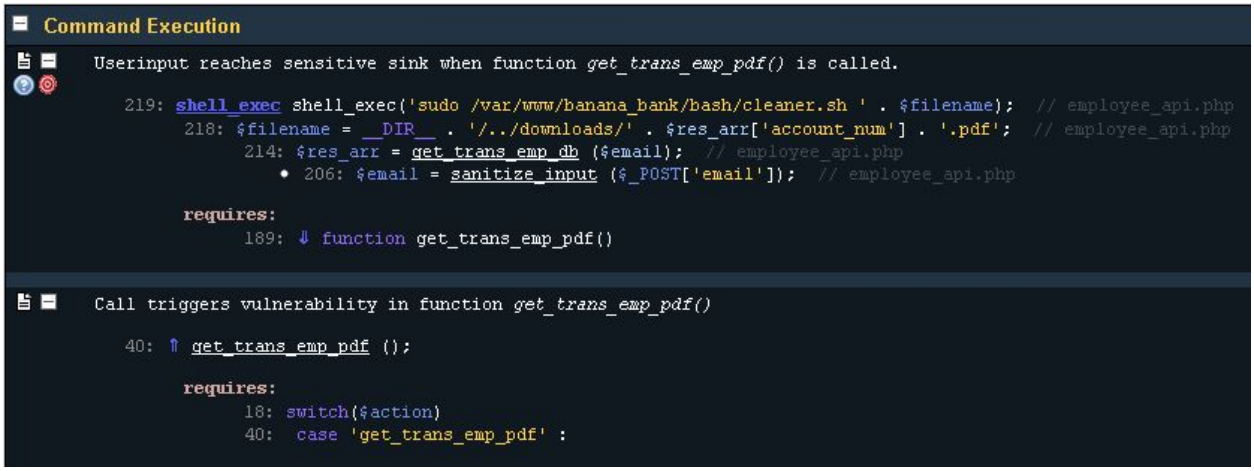
```
if (move_uploaded_file($_FILES['uploadFile']['tmp_name'], $target))
```
(aux_func.php, line 249)

If the upload_file-function reports an error, the parse_file-function will not be called.

But all of this input sanitizing is not enough. It is possible to construct filenames which are also useful commands.

Example: It is possible to create a new file in the 'banana_bank/php'-folder with the following manipulated filename: '| touch xyz .txt'. This can be observed because server misconfiguration which lead to directory listing. With the filename '| rm xyz .txt' you can delete the previous file or any other file in the folder like the serve_request.php. Deleting this file would cause a denial of service. We realized that sudo could be used without providing a password (see command execution 1), so root access is provided.

Command Execution - 3 - get_trans_emp_pdf



The get_trans_emp_pdf-function executes a shell command which is a concatenation of a constant and the $filename variable.

```
shell_exec shell_exec('sudo /var/www/banana_bank/bash/cleaner.sh ' . $filename);
```

The $filename itself is also a concatenation of a constant and the $res_arr['account_num'] variable.

```
$filename = __DIR__ . '/../downloads/' . $res_arr['account_num'] . '.pdf';
```

This variable is filled by the get_trans_emp_db-function.

```
$res_arr = get_trans_emp_db ($email);  // employee_api.php
```

get_trans_emp_db reads the account number from the database, which is of type integer. Therefore it can not cause command execution. The statement to get the information is sanitized, so the database result is trustworthy and therefore also the $filename.

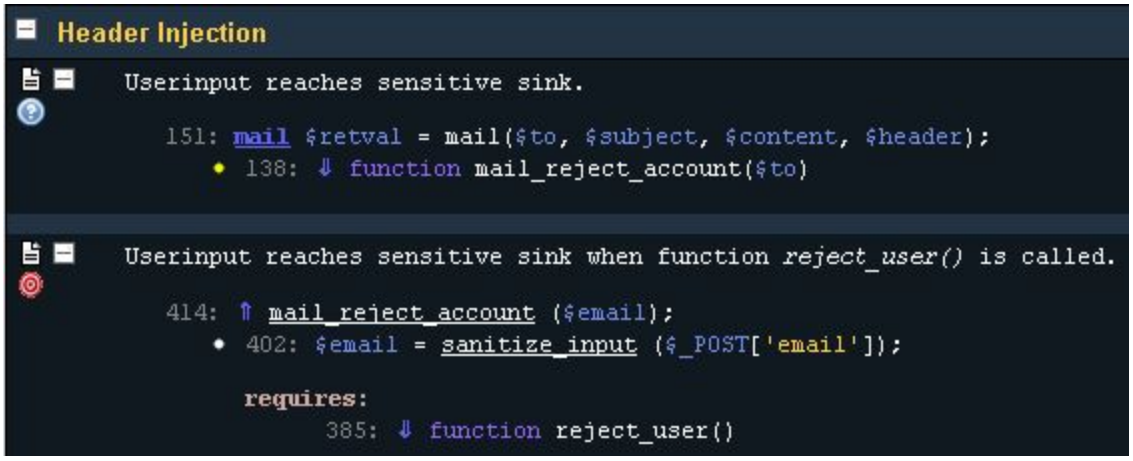
```
$email = mysql_real_escape_string($email);
$query = 'select account_number from BALANCE where email="' . $email . '"';
```

So no vulnerabilities exists. It is a false positive.

File Manipulation

See explanation of File Manipulation of RUN 1.

Header Injection - 1 - mail_reject_account



The mail_reject_account-function uses the php mail-function to send an email to a registered user. Depending on the used mail client, an attacker may cause some damage.

```
mail $retval = mail($to, $subject, $content, $header);
```
(aux_func.php, line 51)

The $subject and $content variable are constants, so they did not need to be sanitized.

```
$subject = 'Registration to Banana bank';
```
(aux_func.php, line 142)

```
$content = 'Dear Madame/Sir, we regret to inform you that your
 registration to Banana bank was not approved.';
```
(aux_func.php, line 144)

The $header variable is an undefined variable. There is a coding error because $header should be $headers. The $headers variable is a concatenated string of constants and a global variable $SYSTEM_EMAIL. The global variable should only be set by administrator so it should be secure to use it.

```
$headers = 'From:' . $SYSTEM_EMAIL . '\r\n';
$headers .= 'MIME-Version: 1.0\r\n';
$headers .= 'Content-Transfer-Encoding: base64\r\n';
$headers .= 'Content-Type: text/html; charset=ISO-8859-1\r\n';
```
(aux_func.php, line 146ff)

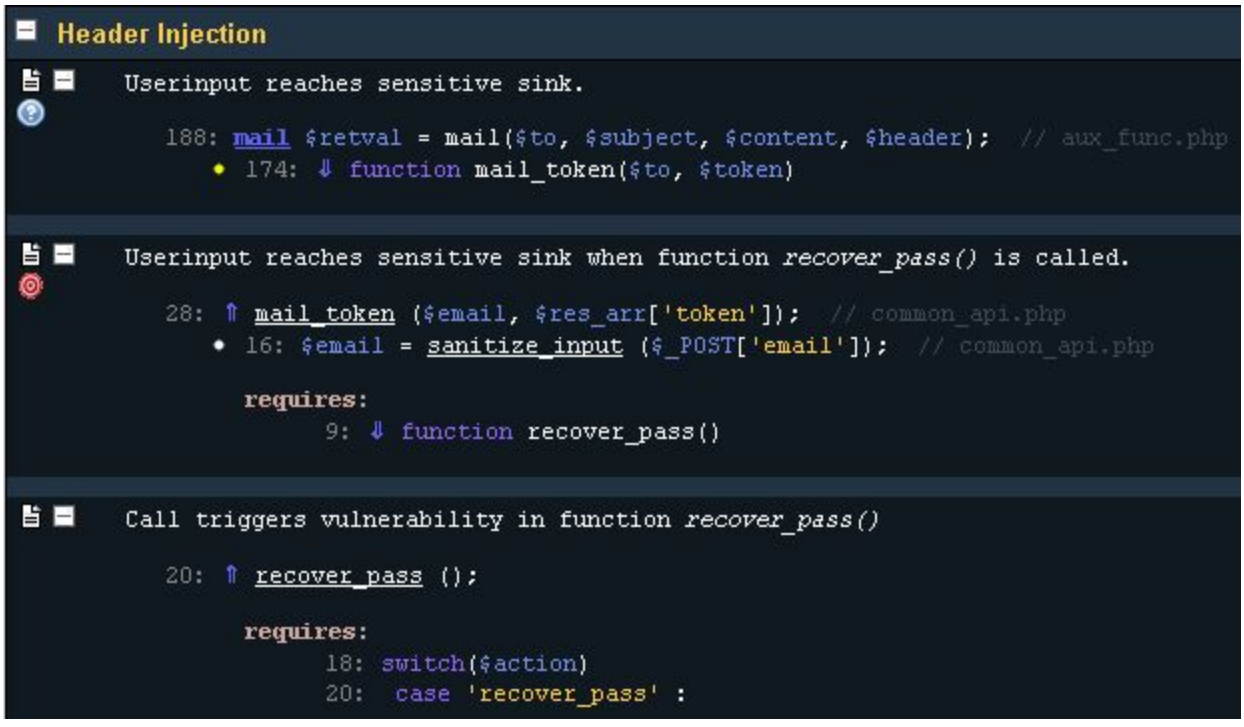
It depends on the security of the input parameter $to. $to is an unsanitized input parameter of the function. So we need to have a closer look at the function stack trace.

Function Stack Trace:

user request
⇒ reject_user
→ $email will be the input parameter $to of mail_reject_account
⇒ sanitize_input($email)
→ offers only little protection against command execution
⇒ **filter_var($email, FILTER_VALIDATE_EMAIL)**
→ format checking
⇒ mail_reject_account

The format of $email is check, so no vulnerability exists. It is a false positive.

Header Injection - 2 - mail_token



The mail_token-function uses the php mail-function to send an email to a registered user. Depending on the used mail client, an attacker may cause some damage.

`$retval = mail($to, $subject, $content, $header);` (aux_func.php, line 188)

The $subject variable is a constant, so they did not need to be sanitized.

```
$subject = 'Password recovery in Banana bank';
```
(aux_func.php, line 178)

The $header variable is an undefined variable. There is a coding error because $header should be $header**s**. The $headers variable is a concatenated string of constants and a global variable $SYSTEM_EMAIL. The global variable should only be set by administrator so it should be secure to use it.

```
$headers = 'From:' . $SYSTEM_EMAIL . '\r\n';
$headers .= 'MIME-Version: 1.0\r\n';
$headers .= 'Content-Transfer-Encoding: base64\r\n';
$headers .= 'Content-Type: text/html; charset=ISO-8859-1\r\n';
```
(aux_func.php, line 183ff)

The $content variable is a concatenated string of a constant and the input parameter $token.

```
$content = 'Dear Madame/Sir, click on the url below in order to change your password:  ';
$content .= 'http://localhost/banana_bank/html/changePass.html?token=' . $token;
```
(aux_func.php, line 180f)

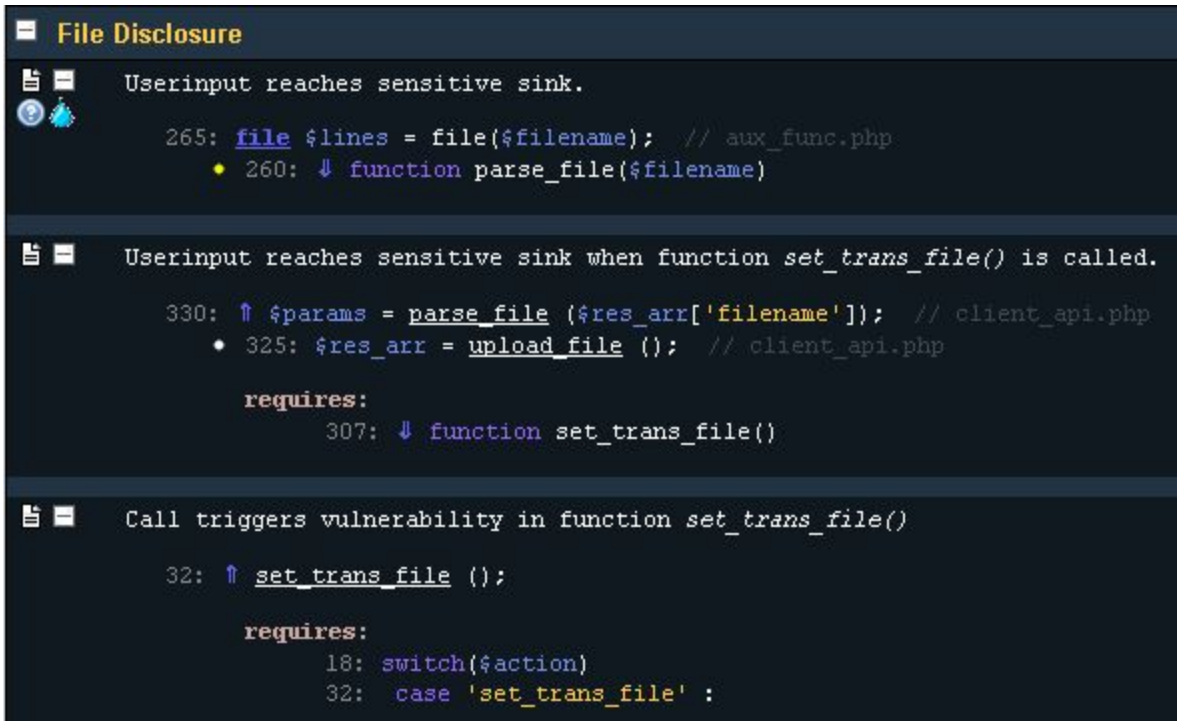
It depends on the security of the input parameter $to and $token. Both are unsanitized input parameters of the function. So we need to have a closer look at the function stack trace.

Function Stack Trace:

user request
- ⇒ recover_pass
- → $email will be the input parameter $to of mail_token
  - ⇒ sanitize_input($email)
  - → offers only little protection against command execution
  - ⇒ **filter_var($email, FILTER_VALIDATE_EMAIL)**
  - → format checking
  - ⇒ recover_pass_db
    - ⇒ **$token = sha1(openssl_random_pseudo_bytes(20))**
    - → did not depend on user input
  - ⇒ mail_token

The format of $email is check and $token did not depend on user input, so no vulnerability exists. It is a false positive.

File Disclosure - 1 - parse_file



The parse_file-function uses the php file-function to read the whole file into an array. The input parameter $filename is directly used as input for the file-function. If the $filename isn't a valid filename or url then no file will be found. This may cause logical faults.

```
$lines = file($filename);
```
(aux_func.php, line 265)

The parse_file-function is only called by the set_trans_file-function.

```
$params = parse_file ($res_arr['filename']);
```

The value of $res_arr['filename'] is set by the upload_file-function.

```
$res_arr = upload_file ();
```
(
```
return array('status' => true, 'filename' => $target);
```
)

The $target variable is a concatenation of a constant and the $name variable. The $name variable is sanitized by the sanitize_input-function and the php $_FILES functionality.

```
$name = sanitize_input($_FILES['uploadFile']['name']);
$target = '/var/www/banana_bank/uploads/' . $name;
```
(aux_func.php, line 246f)

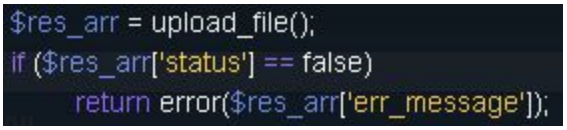
If the $target variable isn't a valid filename, the $move_uploaded_file would fail and the upload_file-function would report an error.

```
if (move_uploaded_file($_FILES['uploadFile']['tmp_name'], $target))
```
(aux_func.php, line 249)

```
return array('status' => false,
        'err_message' => 'Whoops, something went wrong while trying to upload the file');
```

(aux_func.php, line 252)

If the upload_file-function reports an error, the parse_file-function will not be called.



(client_api.php, line 324)

As a conclusion we could say that there is no vulnerability because it the application logic ensures that the input parameter $filename is a valid filepath.

Reverse Engineering - C++ Program

The only function of the batch program seems to be to output the content of the batch file on stdout. This output is then read from the PHP application and further processed. All input sanitizing is done in the PHP part.

We used several tools to analyze the binary program used for batch processing which is called ‘set_trans_file’. A first naive approach was to search for hard coded passwords and connection data with the ‘strings’ utility of the GNU binutils package. This didn’t seem to reveal any such information.

Normal execution of the program showed the error message “Unable to open input file!!” which can be seen in the lower end of the strings figure. This lead to the conclusion that an input file is needed. Creating an empty file and calling the program with the file name of this empty file showed a new error message “Unable to open file to write” also visible in the strings output.

Using the strace utility to show system calls as in ‘`strace ./set_trans_file empty.txt`’ revealed amongst others a call of the ‘`open`’ syscall with the argument ‘`empty.txt`’ which succeeds and another call to ‘`open`’ with the parameter ‘`/var/www/banana_bank/uploads/temp.txt`’ which fails. After creating the directory ‘`/var/www/banana_bank/uploads`’ this call succeeds. Another execution with strace shows that the program now exits with a return code of 0.

However looking at the full strace output we can see a confusing series of system calls:

1. open empty.txt
2. open /var/www/banana_bank/uploads/temp.txt
3. read all lines from empty.txt
4. close empty.txt
5. write not-empty lines to /var/www/banana_bank/uploads/temp.txt
6. close /var/www/banana_bank/uploads/temp.txt

7. unlink empty.txt
8. rename /var/www/banana_bank/uploads/temp.txt to empty.txt
9. open empty.txt
10. read all lines form empty.txt
11. print every line separately
12. print two spaces

A look at the symbol table using ‘readelf’ also from the GNU binutils and filtering the output with ‘grep’ (`readelf -s set_trans_file | grep -ve UND | grep FUNC`) shows us that there is a function called ‘clear_lines’ which could be responsible for the removal of empty lines we see in the previous list of syscalls.

```
samurai@samurai-wtf:Downloads$ readelf -s set_trans_file | grep -ve UND | grep FUNC
    31: 080485d0     0 FUNC    LOCAL  DEFAULT   13 __do_global_dtors_aux
    34: 08048630     0 FUNC    LOCAL  DEFAULT   13 frame_dummy
    39: 08048ac0     0 FUNC    LOCAL  DEFAULT   13 __do_global_ctors_aux
    45: 08048ab0     2 FUNC    GLOBAL DEFAULT   13 __libc_csu_fini
    47: 08048ab2     0 FUNC    GLOBAL HIDDEN    13 __i686.get_pc_thunk.bx
    55: 08048aec     0 FUNC    GLOBAL DEFAULT   14 _fini
    66: 08048733   186 FUNC    GLOBAL DEFAULT   13 read_file
    68: 08048a40    97 FUNC    GLOBAL DEFAULT   13 __libc_csu_init
    71: 080487ed   433 FUNC    GLOBAL DEFAULT   13 clear_lines
    72: 080485a0     0 FUNC    GLOBAL DEFAULT   13 _start
    73: 08048654   223 FUNC    GLOBAL DEFAULT   13 tokenize
    77: 0804899e   162 FUNC    GLOBAL DEFAULT   13 main
    83: 08048470     0 FUNC    GLOBAL DEFAULT   11 _init
```

The string ‘`%s %s %s`’ gives us a hint on how the input file might have to look like. Judging by the symbol table and strings output the only insecure function used is strcpy, however tests with varying input lengths up to the maximum of 65535 showed no segmentation fault which would be a hint for a buffer overflow. Therefore we deducted that there is no exploitable buffer overflow.

Searching the callsite of the binary inside the php application showed us that the showed us that only the output of the binary - what is printed to stdout - is used in the php application, but the php application is the only component that knows the database connection data and establishes database connections.

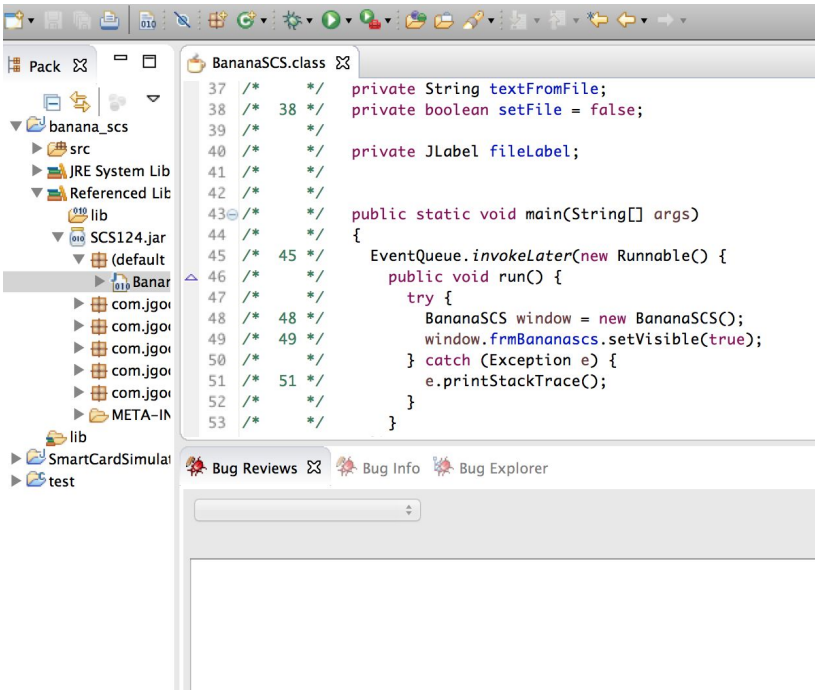
The parts of the php application that make use of the binary have been identified using the command ‘`find . -name "*.php" -exec grep "set_trans_file\b" --with-filename --color --line-number {} +`’ which shows us that the function ‘`parse_file`’ in the file ‘`php/aux_func.php`’ calls the binary in line 270. Checking the code there reveals that the php program only reads it’s output and does the database communication completely in php.

```
samurai@samurai-wtf:Code$ find . -name "*.php" -exec grep "set_trans_file\b" --with-filename --line-number --color {} +
./php/serve_requests.php:16:    $action = 'set_trans_file';
./php/serve_requests.php:32:    case 'set_trans_file':  return set_trans_file();
./php/client_api.php:307:function set_trans_file() {
./php/aux_func.php:270:    $handle = popen('/var/www/banana_bank/exe/set_trans_file ' . $filename, 'r');
```

Reverse Engineering - Java Program

We did the reverse engineering of SCS jar using JD-Eclipse Plugin (0.1.5) and found that the java application uses a hardcoded secret key to generate the tan number. The secret key is stored in a variable named "secret" at line number 32 of class BananaSCS.class .
However we also found that this secret key is not really private as one client can download the SCS jar of another client thus posing a great threat on security.

Analysing the java code further using FindBug revealed no bugs.



The reverse engineering also revealed that the SCS jar uses a random number from to 0 to 1001 to generate the hash. The php server then looks for all possible hashes with strings containing values from 0 to 1001. This implies that more than one tan numbers are possible for a transaction.

Additional Info

Database connection data

Searching for the database connection data and passwords showed us that banana bank uses the phpsec library by OWASP to store encrypted passwords and user in their configuration.

We could identify a flaw in their usage of phpsec's string encryption: They didn't provide a new password and initialization vector and so anybody that downloads the library knows these secrets and can decrypt the encrypted strings if he can access them so the security gained by the encryption is only week if there is a gain at all.

Another thing we realized is that the application stores the root user name of the database as well as the user the application uses to access the database. As both encrypted strings are identical we also got the idea that there is no sophisticated db user for the application.

By writing a simple php script to decrypts the encrypted strings we could tell that the application uses the root account to access the database and we were also able to find out what the root password is.

php script

<?php
require_once __DIR__ . '/phpsec/crypto/confidentialstring.php';
 	
$DB_USER =
phpsec\confidentialString(':sENSt7jtm5WBRy14P95atM8qa8ttFt0COQwkvyIKca8=');
$DB_PASS =
phpsec\confidentialString(':ZaGjIXgNDKH668WPPZEMbSjFECBewKSjIYrykS0rMGk=');
$ROOT_PASS =
phpsec\confidentialString(':ZaGjIXgNDKH668WPPZEMbSjFECBewKSjIYrykS0rMGk=');
$DB_NAME =
phpsec\confidentialString(':1cLIvEkzWViP7sz5RdTDtzBuU5rjBwK47X+rmtYUJFY=');
 	
echo 'DB_USER: ' . $DB_USER . '
';
echo 'DB_PASS: ' . $DB_PASS . '
';
echo 'ROOT_PASS: ' . $ROOT_PASS . '
';
echo 'DB_NAME: ' . $DB_NAME . '
';
 	
?>

output

DB_USER: root
DB_PASS: alpha12
ROOT_PASS: alpha12
DB_NAME: my_bank

Manual code review

While reading the code to understand the architecture and business logic of the application we found some discrepancies.

Data type of amounts

While the table for accounts stores the balance as a data type `` `balance` DOUBLE ( 20 , 2 ) NOT NULL `` whereas the table for transaction uses `` `amount` MEDIUMINT ( 8 ) UNSIGNED NOT NULL ``. This discrepancy is visible again in line 276 in the function '`parse_file`' where float delimiters are not part of the valid characters but later on the same data get converted to float (line 470, function '`set_trans_file_db`') and used in the database statements for both, updating the accounts and creating the transfer.

Reject of employee not possible

Observation:

We found that is it not possible to reject a new employee. If you try to do it, an error is shown.

Discovery:

1. create a new employee
2. login with an existing employee
3. try to reject the new employee
4. the following error message is shown