ASSIGNMENT 3

ENGINEERING

Neilos Kotsiopoulos

neko@itu.dk

1) Problem 1: Security Requirements

Part 1: Perform a threat analysis and a harm analysis of PayBud

1. What are the assets of PayBud?
   The primary assets of PayBud are its user's money stored in the accounts
2. What are the threats of concern? Their motivation? Capabilities?
   The major threats are from cybercriminals seeking financial gain and potentially hacktivists who may wish to disrupt the financial system. These individuals or groups can range from being highly skilled and funded to moderately skilled with limited resources.
3. For each asset, what are the possible harms? Does this affect confidentiality, integrity or availability of the asset? (see harm triples)
   - The possible harms to the assets include unauthorized transactions leading to financial loss (affecting the integrity of the user's account balance)
   - DDoS attacks could disrupt services making transactions and account access unavailable (affecting the availability of the system)

Part 2: Write security goals for PayBud based on your analysis. These have the form "the system should {prevent, detect} action on asset."

1. The system should prevent unauthorized access and transactions on user accounts
2. The system should detect and mitigate any attempts to disrupt the availability of its services

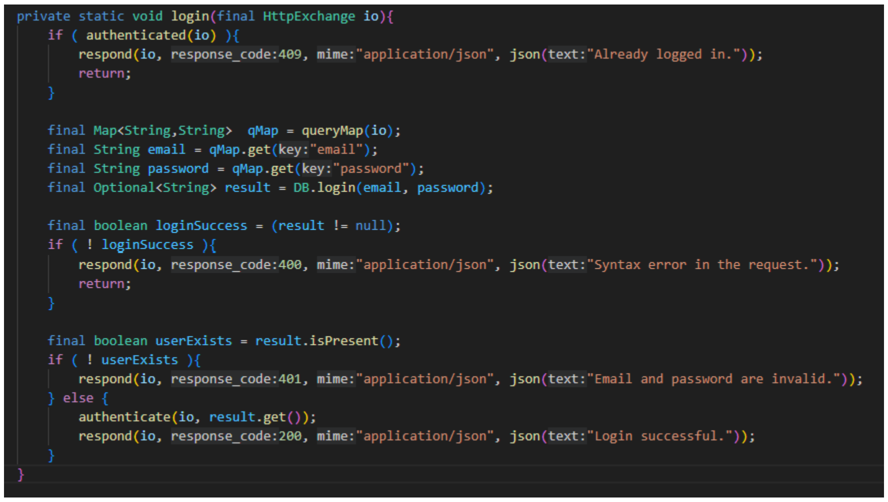
Part 3: Write security requirements that are needed to satisfy these security goals. These are constraints on functional requirements

1. The system requires two-factor authentication (2FA) when users log in to protect against unauthorized access
2. The system should enforce a rule that if the sum of credit card transactions exceeds a certain limit (e.g., 5000kr) within a specified period (e.g., 24 hours), additional verification is required to prevent unauthorized transactions

2) Problem 2: Audit

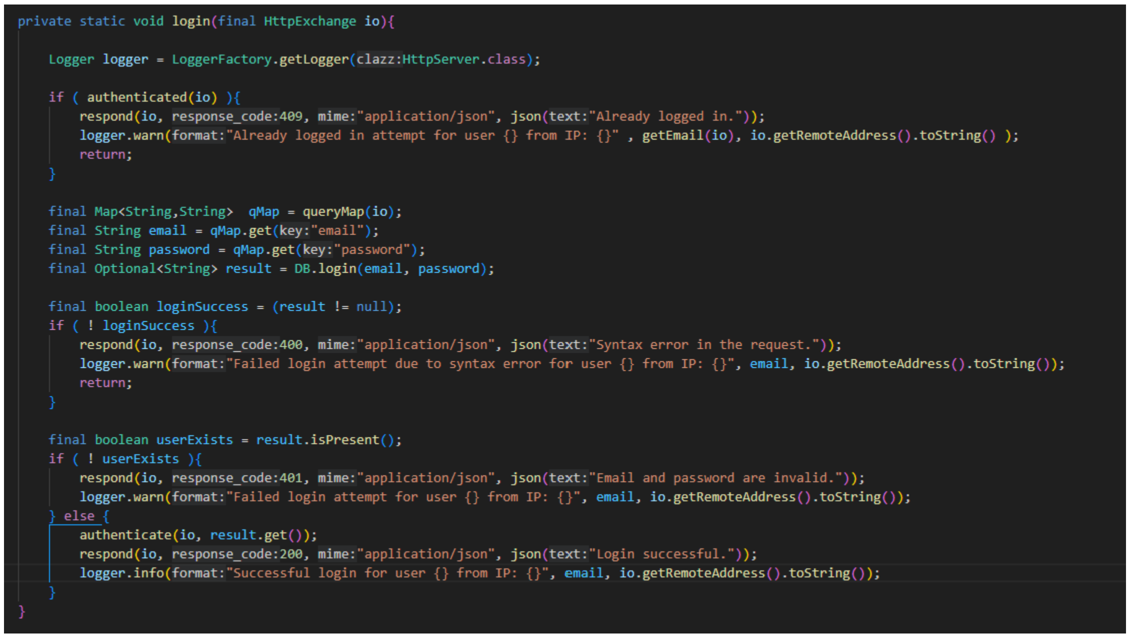
As the assignment suggests, first I had to determine where in the code to add the logging statements and in which security relevant events

- In order to monitor potential brute-force logging in attacks, I would add a logging statement for each successful / unsuccessful logging attempt of each user, at the following piece of code:



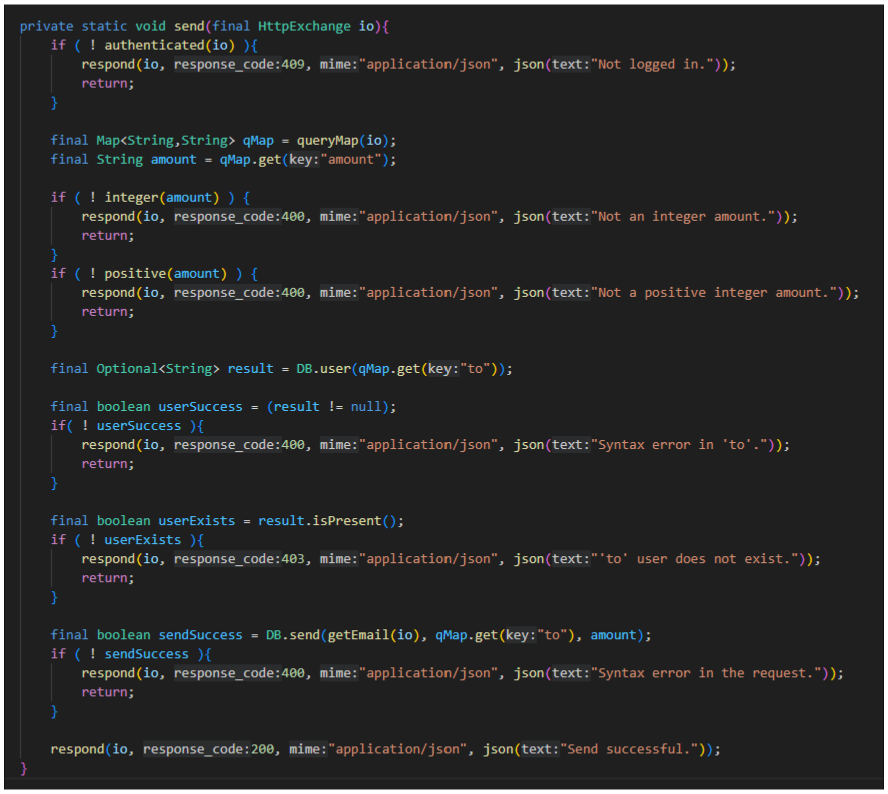
```
private static void login(final HttpExchange io){
    if ( authenticated(io) ){
        respond(io, response_code:409, mime:"application/json", json(text:"Already logged in."));
        return;
    }

    final Map<String,String>  qMap = queryMap(io);
    final String email = qMap.get(key:"email");
    final String password = qMap.get(key:"password");
    final Optional<String> result = DB.login(email, password);

    final boolean loginSuccess = (result != null);
    if ( ! loginSuccess ){
        respond(io, response_code:400, mime:"application/json", json(text:"Syntax error in the request."));
        return;
    }

    final boolean userExists = result.isPresent();
    if ( ! userExists ){
        respond(io, response_code:401, mime:"application/json", json(text:"Email and password are invalid."));
    } else {
        authenticate(io, result.get());
        respond(io, response_code:200, mime:"application/json", json(text:"Login successful."));
    }
}
```

Changing the Code to:



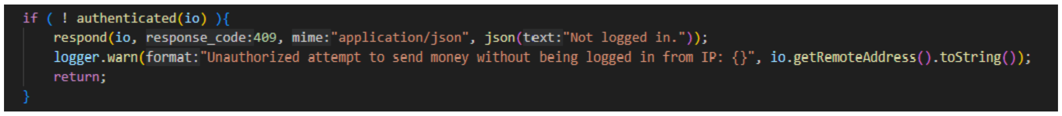
```
private static void login(final HttpExchange io){

    Logger logger = LoggerFactory.getLogger(clazz:HttpServer.class);

    if ( authenticated(io) ){
        respond(io, response_code:409, mime:"application/json", json(text:"Already logged in."));
        logger.warn(format:"Already logged in attempt for user {} from IP: {}" , getEmail(io), io.getRemoteAddress().toString() );
        return;
    }

    final Map<String,String>  qMap = queryMap(io);
    final String email = qMap.get(key:"email");
    final String password = qMap.get(key:"password");
    final Optional<String> result = DB.login(email, password);

    final boolean loginSuccess = (result != null);
    if ( ! loginSuccess ){
        respond(io, response_code:400, mime:"application/json", json(text:"Syntax error in the request."));
        logger.warn(format:"Failed login attempt due to syntax error for user {} from IP: {}", email, io.getRemoteAddress().toString());
        return;
    }

    final boolean userExists = result.isPresent();
    if ( ! userExists ){
        respond(io, response_code:401, mime:"application/json", json(text:"Email and password are invalid."));
        logger.warn(format:"Failed login attempt for user {} from IP: {}", email, io.getRemoteAddress().toString());
    } else {
        authenticate(io, result.get());
        respond(io, response_code:200, mime:"application/json", json(text:"Login successful."));
        logger.info(format:"Successful login for user {} from IP: {}", email, io.getRemoteAddress().toString());
    }
}
```

-In order to monitor attempts regarding transactions (either from logged in or not logged in user attempts) the following piece of code need to be changed:



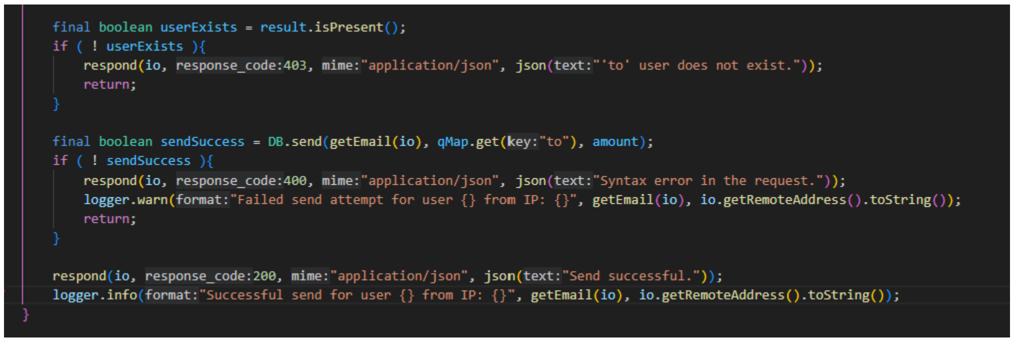
```
private static void send(final HttpExchange io){
    if ( ! authenticated(io) ){
        respond(io, response_code:409, mime:"application/json", json(text:"Not logged in."));
        return;
    }

    final Map<String,String> qMap = queryMap(io);
    final String amount = qMap.get(key:"amount");

    if ( ! integer(amount) ) {
        respond(io, response_code:400, mime:"application/json", json(text:"Not an integer amount."));
        return;
    }
    if ( ! positive(amount) ) {
        respond(io, response_code:400, mime:"application/json", json(text:"Not a positive integer amount."));
        return;
    }

    final Optional<String> result = DB.user(qMap.get(key:"to"));

    final boolean userSuccess = (result != null);
    if( ! userSuccess ){
        respond(io, response_code:400, mime:"application/json", json(text:"Syntax error in 'to'."));
        return;
    }

    final boolean userExists = result.isPresent();
    if ( ! userExists ){
        respond(io, response_code:403, mime:"application/json", json(text:"'to' user does not exist."));
        return;
    }

    final boolean sendSuccess = DB.send(getEmail(io), qMap.get(key:"to"), amount);
    if ( ! sendSuccess ){
        respond(io, response_code:400, mime:"application/json", json(text:"Syntax error in the request."));
        return;
    }

    respond(io, response_code:200, mime:"application/json", json(text:"Send successful."));
}
```

that has been changed to:



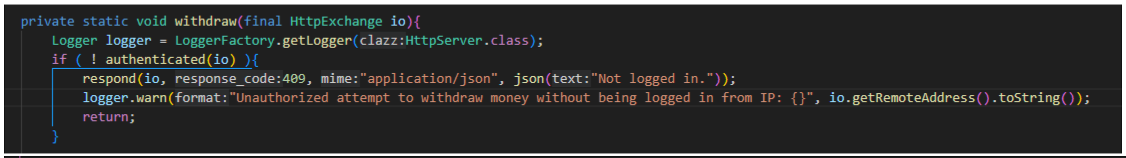
```
if ( ! authenticated(io) ){
    respond(io, response_code:409, mime:"application/json", json(text:"Not logged in."));
    logger.warn(format:"Unauthorized attempt to send money without being logged in from IP: {}", io.getRemoteAddress().toString());
    return;
}
```

and

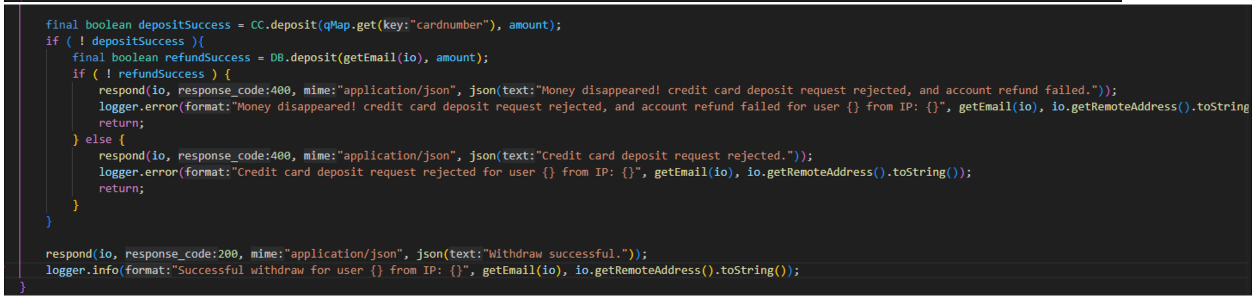


```
    final boolean userExists = result.isPresent();
    if ( ! userExists ){
        respond(io, response_code:403, mime:"application/json", json(text:"'to' user does not exist."));
        return;
    }

    final boolean sendSuccess = DB.send(getEmail(io), qMap.get(key:"to"), amount);
    if ( ! sendSuccess ){
        respond(io, response_code:400, mime:"application/json", json(text:"Syntax error in the request."));
        logger.warn(format:"Failed send attempt for user {} from IP: {}", getEmail(io), io.getRemoteAddress().toString());
        return;
    }

    respond(io, response_code:200, mime:"application/json", json(text:"Send successful."));
    logger.info(format:"Successful send for user {} from IP: {}", getEmail(io), io.getRemoteAddress().toString());
}
```

-Same applies in order to monitor withdrawal attempts



```
private static void withdraw(final HttpExchange io){
    Logger logger = LoggerFactory.getLogger(clazz:HttpServer.class);
    if ( ! authenticated(io) ){
        respond(io, response_code:409, mime:"application/json", json(text:"Not logged in."));
        logger.warn(format:"Unauthorized attempt to withdraw money without being logged in from IP: {}", io.getRemoteAddress().toString());
        return;
    }
```

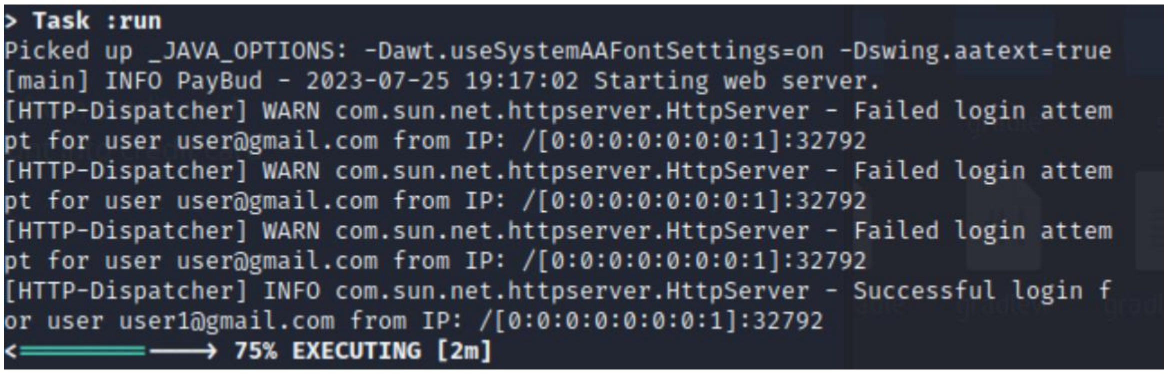


```
    final boolean depositSuccess = CC.deposit(qMap.get(key:"cardnumber"), amount);
    if ( ! depositSuccess ){
        final boolean refundSuccess = DB.deposit(getEmail(io), amount);
        if ( ! refundSuccess ) {
            respond(io, response_code:400, mime:"application/json", json(text:"Money disappeared! credit card deposit request rejected, and account refund failed."));
            logger.error(format:"Money disappeared! credit card deposit request rejected, and account refund failed for user {} from IP: {}", getEmail(io), io.getRemoteAddress().toString
            return;
        } else {
            respond(io, response_code:400, mime:"application/json", json(text:"Credit card deposit request rejected."));
            logger.error(format:"Credit card deposit request rejected for user {} from IP: {}", getEmail(io), io.getRemoteAddress().toString());
            return;
        }
    }

    respond(io, response_code:200, mime:"application/json", json(text:"Withdraw successful."));
    logger.info(format:"Successful withdraw for user {} from IP: {}", getEmail(io), io.getRemoteAddress().toString());
}
```

Which security-relevant events are you logging?

1. Unauthorized attempts to withdraw money, which signify potential breaches or abuse of your system. This includes the IP address from where these attempts are made
2. Transactions where money disappears due to credit card deposit failure after successful withdrawal from the account, or when a credit card deposit request is rejected.

Log example snippet below:



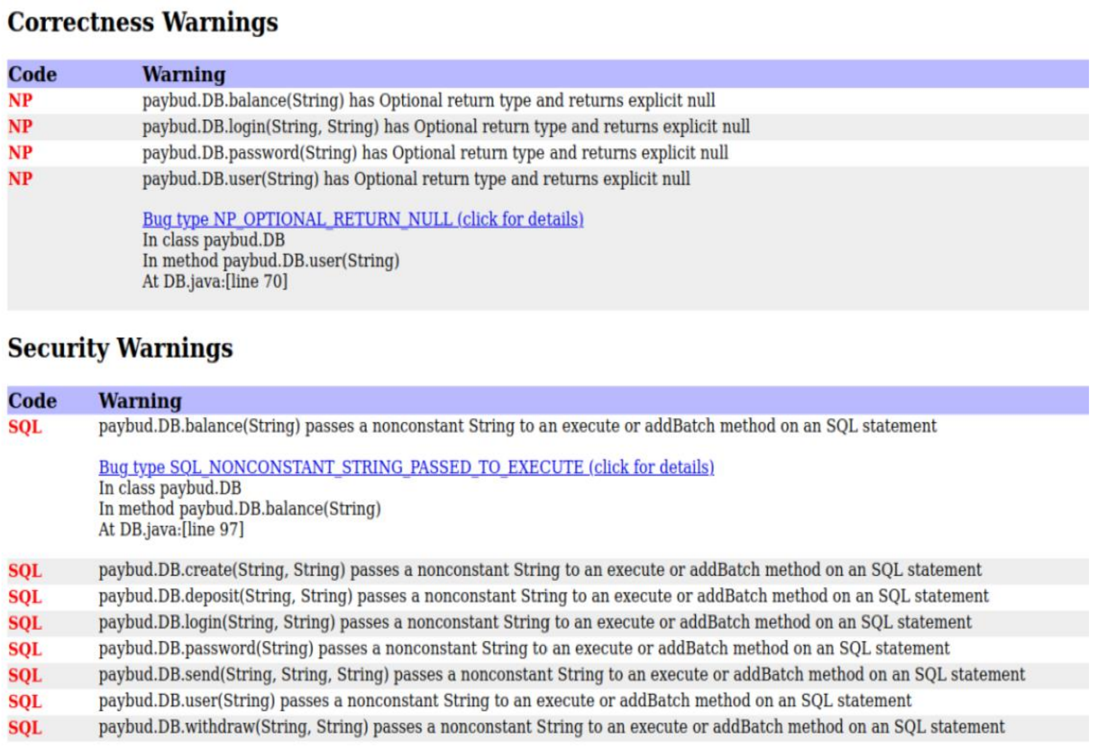
```
> Task :run
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
[main] INFO PayBud - 2023-07-25 19:17:02 Starting web server.
[HTTP-Dispatcher] WARN com.sun.net.httpserver.HttpServer - Failed login attem
pt for user user@gmail.com from IP: /[0:0:0:0:0:0:0:1]:32792
[HTTP-Dispatcher] WARN com.sun.net.httpserver.HttpServer - Failed login attem
pt for user user@gmail.com from IP: /[0:0:0:0:0:0:0:1]:32792
[HTTP-Dispatcher] WARN com.sun.net.httpserver.HttpServer - Failed login attem
pt for user user@gmail.com from IP: /[0:0:0:0:0:0:0:1]:32792
[HTTP-Dispatcher] INFO com.sun.net.httpserver.HttpServer - Successful login f
or user user1@gmail.com from IP: /[0:0:0:0:0:0:0:1]:32792
<===========---> 75% EXECUTING [2m]
```

3) Problem 3: Attack & SAST

Part 1: What vulnerability did spotbugs find? In which files/lines?

Spotbugs identified two types of volnerabilities:

1. 4 instances of NullPoinerExceptions in Boolean methods in paybud.DB, in lines 52,70,88,106
2. 8 instances of noncostanct String passes to an SQL statements, in lines: 26, 43, 61, 79, 97, 117, 134, 147



Correctness Warnings

Code	Warning
NP	paybud.DB.balance(String) has Optional return type and returns explicit null
NP	paybud.DB.login(String, String) has Optional return type and returns explicit null
NP	paybud.DB.password(String) has Optional return type and returns explicit null
NP	paybud.DB.user(String) has Optional return type and returns explicit null
	Bug type NP_OPTIONAL_RETURN_NULL (click for details)
	In class paybud.DB
	In method paybud.DB.user(String)
	At DB.java:[line 70]

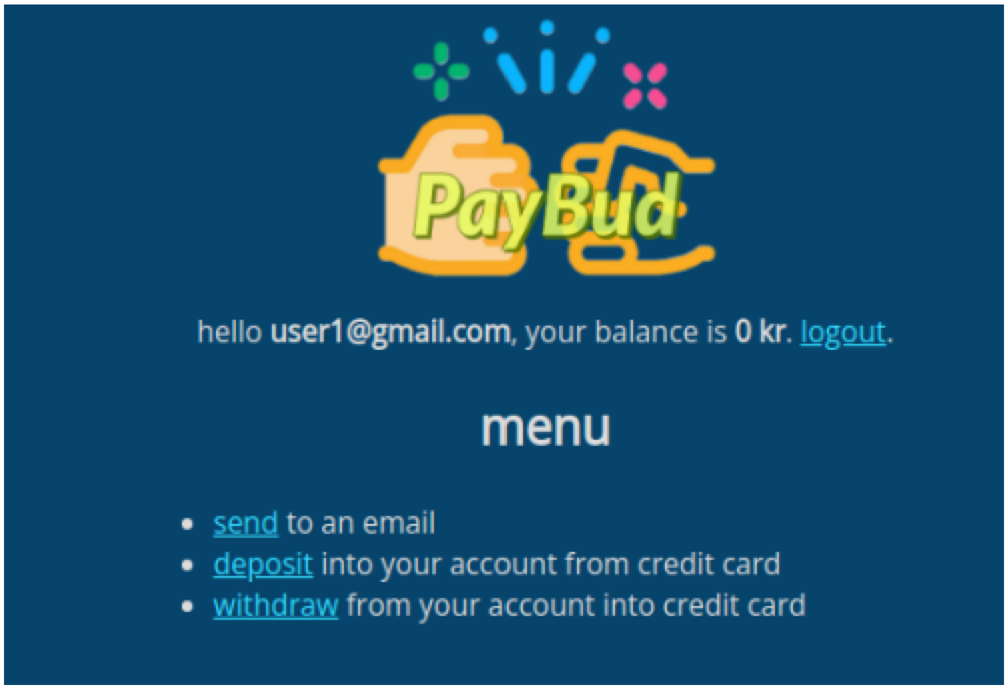
Security Warnings

Code	Warning
SQL	paybud.DB.balance(String) passes a nonconstant String to an execute or addBatch method on an SQL statement
	Bug type SQL_NONCONSTANT_STRING_PASSED_TO_EXECUTE (click for details)
	In class paybud.DB
	In method paybud.DB.balance(String)
	At DB.java:[line 97]
SQL	paybud.DB.create(String, String) passes a nonconstant String to an execute or addBatch method on an SQL statement
SQL	paybud.DB.deposit(String, String) passes a nonconstant String to an execute or addBatch method on an SQL statement
SQL	paybud.DB.login(String, String) passes a nonconstant String to an execute or addBatch method on an SQL statement
SQL	paybud.DB.password(String) passes a nonconstant String to an execute or addBatch method on an SQL statement
SQL	paybud.DB.send(String, String, String) passes a nonconstant String to an execute or addBatch method on an SQL statement
SQL	paybud.DB.user(String) passes a nonconstant String to an execute or addBatch method on an SQL statement
SQL	paybud.DB.withdraw(String, String) passes a nonconstant String to an execute or addBatch method on an SQL statement

Part 2: Demonstrate an exploit using this vulnerability

The nonconstant warning, can be identified as potential SQL injection attack vulnerability

3. I tried to login using an existing username name “user1” and different SQL commands as passwords – I managed to logged in by using < ‘ or 1=1;-- >as password



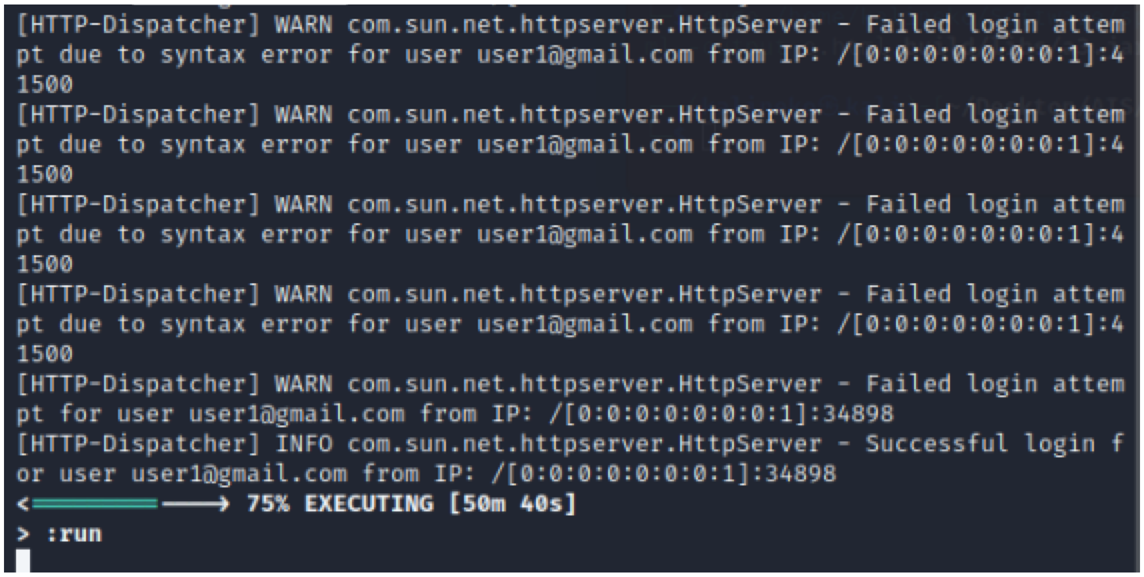
hello **user1@gmail.com**, your balance is **0 kr**. logout.

menu

- send to an email
- deposit into your account from credit card
- withdraw from your account into credit card

Part 3: Show how the logging you implemented above reveals the exploit

4. Below is a screenshot of the logging statements identify the vulnerability



```
[HTTP-Dispatcher] WARN com.sun.net.httpserver.HttpServer - Failed login attem
pt due to syntax error for user user1@gmail.com from IP: /[0:0:0:0:0:0:0:1]:4
1500
[HTTP-Dispatcher] WARN com.sun.net.httpserver.HttpServer - Failed login attem
pt due to syntax error for user user1@gmail.com from IP: /[0:0:0:0:0:0:0:1]:4
1500
[HTTP-Dispatcher] WARN com.sun.net.httpserver.HttpServer - Failed login attem
pt due to syntax error for user user1@gmail.com from IP: /[0:0:0:0:0:0:0:1]:4
1500
[HTTP-Dispatcher] WARN com.sun.net.httpserver.HttpServer - Failed login attem
pt due to syntax error for user user1@gmail.com from IP: /[0:0:0:0:0:0:0:1]:4
1500
[HTTP-Dispatcher] WARN com.sun.net.httpserver.HttpServer - Failed login attem
pt for user user1@gmail.com from IP: /[0:0:0:0:0:0:0:1]:34898
[HTTP-Dispatcher] INFO com.sun.net.httpserver.HttpServer - Successful login f
or user user1@gmail.com from IP: /[0:0:0:0:0:0:0:1]:34898
<━━━━━━━━━━───→ 75% EXECUTING [50m 40s]
> :run
```

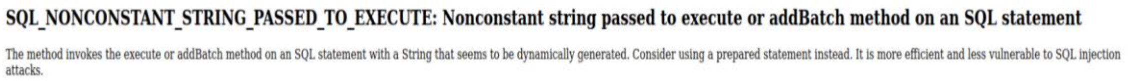
Part 4: Which security requirements does this exploit violate?

The above exploit exposes security vulnerabilities that violate the confidentiality and integrity of the asset, which in this case is the money

-Only account owners should access their accounts and conduct transactions

Part 5: What is spotbugs's proposed fix? Why does that fix the vulnerability?

Spotbugs proposes the following solution for the above vulnerability:



SQL_NONCONSTANT_STRING_PASSED_TO_EXECUTE: Nonconstant string passed to execute or addBatch method on an SQL statement

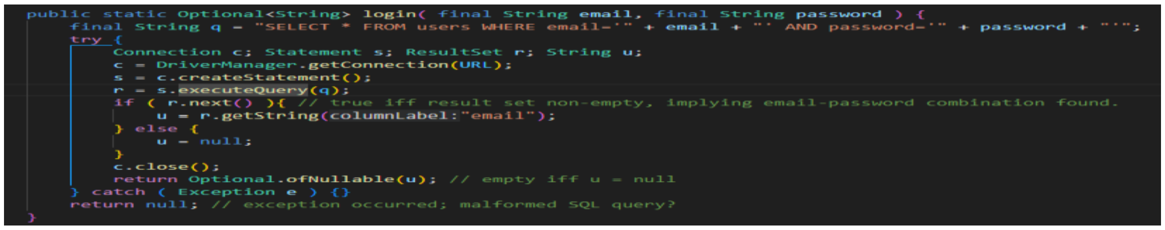
The method invokes the execute or addBatch method on an SQL statement with a String that seems to be dynamically generated. Consider using a prepared statement instead. It is more efficient and less vulnerable to SQL injection attacks.

Using a Prepared statement instead will make the SQL injection impossible.

As for the optional return null, Boolean statements must only return either true or false, and in case of null, false.

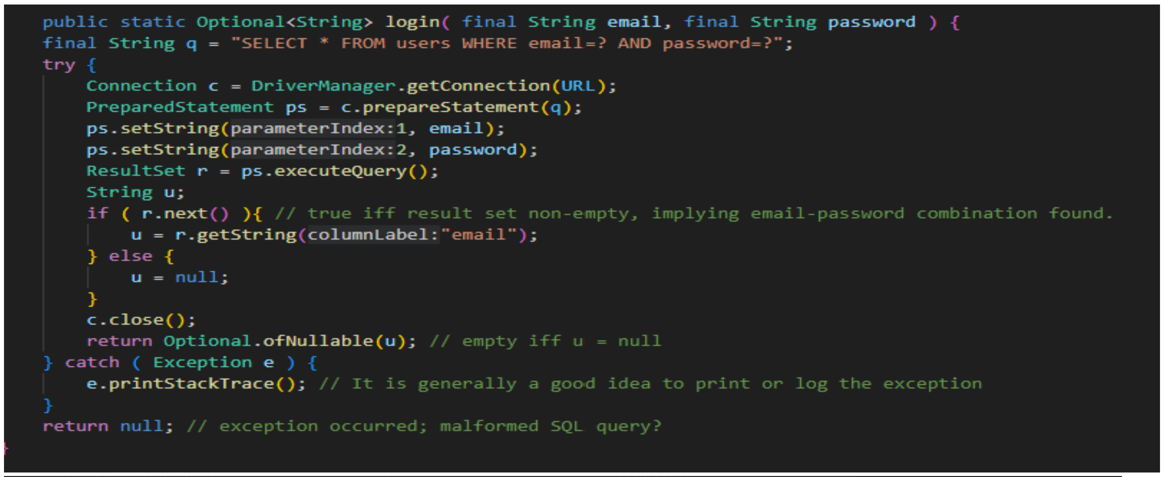
Part 6: Do the proposed fix. Explain (for one instance of the vulnerability) how you modified the code

Firstly, let's check the part of the code that is exposed according to spotbugs report (i.e. line 43 at paybud.DB.login)

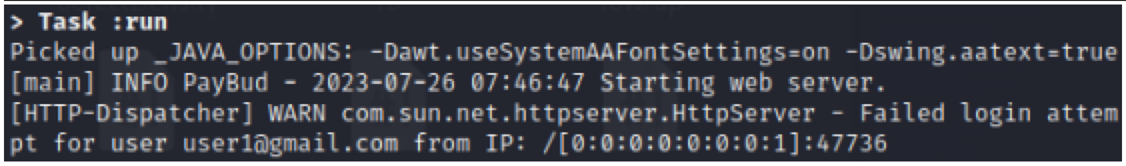


```
public static Optional<String> login( final String email, final String password ) {
    final String q = "SELECT * FROM users WHERE email='" + email + "' AND password='" + password + "'";
    try {
        Connection c; Statement s; ResultSet r; String u;
        c = DriverManager.getConnection(URL);
        s = c.createStatement();
        r = s.executeQuery(q);
        if ( r.next() ){ // true iff result set non-empty, implying email-password combination found.
            u = r.getString(columnLabel:"email");
        } else {
            u = null;
        }
        c.close();
        return Optional.ofNullable(u); // empty iff u = null
    } catch ( Exception e ) {}
    return null; // exception occurred; malformed SQL query?
}
```

The following snippet shows how the code should be changed to prevent SQL injection attacks through prepared statements



```java
public static Optional<String> login( final String email, final String password ) {
final String q = "SELECT * FROM users WHERE email=? AND password=?";
try {
    Connection c = DriverManager.getConnection(URL);
    PreparedStatement ps = c.prepareStatement(q);
    ps.setString(parameterIndex:1, email);
    ps.setString(parameterIndex:2, password);
    ResultSet r = ps.executeQuery();
    String u;
    if ( r.next() ){ // true iff result set non-empty, implying email-password combination found.
        u = r.getString(columnLabel:"email");
    } else {
        u = null;
    }
    c.close();
    return Optional.ofNullable(u); // empty iff u = null
} catch ( Exception e ) {
    e.printStackTrace(); // It is generally a good idea to print or log the exception
}
return null; // exception occurred; malformed SQL query?
```

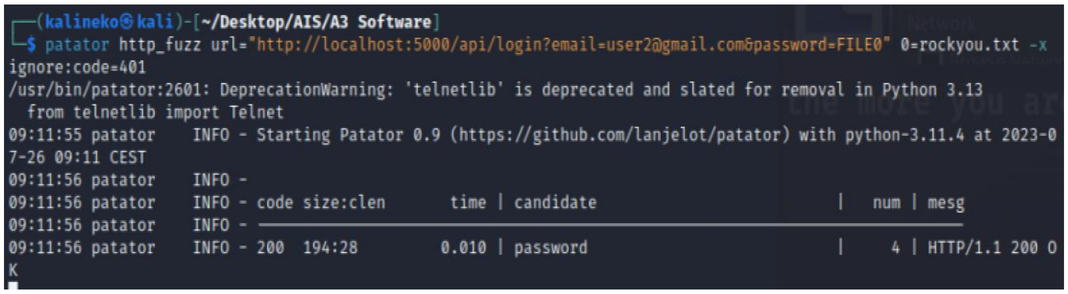


```
> Task :run
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
[main] INFO PayBud - 2023-07-26 07:46:47 Starting web server.
[HTTP-Dispatcher] WARN com.sun.net.httpserver.HttpServer - Failed login attem
pt for user user1@gmail.com from IP: /[0:0:0:0:0:0:0:1]:47736
```

4) Problem 4: Attack, Online

Part 1: Demonstrate an exploit using this vulnerability, by following the patator instructions provided alongside this assignment. Explain every step of your attack; include each command, explain what it does, explain why you are doing it (i.e. to what end), display the output that shows that your attack is successful.

5. First I installed the patator patch as per notes & Slack instructions
6. I created a new user, using a password from the rockyou.txt list
7. Then I ran the command < PYTHONWARNINGS=ignore patator http_fuzz url="http://localhost:5000/api/login?email=user2@gmail.com&password=FILE0" 0=/home/kalineko/Desktop/AIS/A3/Software/rockyou.txt -x ignore:code=401 >



```
┌──(kalineko㉿kali)-[~/Desktop/AIS/A3 Software]
└─$ patator http_fuzz url="http://localhost:5000/api/login?email=user2@gmail.com&password=FILE0" 0=rockyou.txt -x
ignore:code=401
/usr/bin/patator:2601: DeprecationWarning: 'telnetlib' is deprecated and slated for removal in Python 3.13
  from telnetlib import Telnet
09:11:55 patator    INFO - Starting Patator 0.9 (https://github.com/lanjelot/patator) with python-3.11.4 at 2023-0
7-26 09:11 CEST
09:11:56 patator    INFO -
09:11:56 patator    INFO - code size:clen       time | candidate                          |   num | mesg
09:11:56 patator    INFO - -----------------------------------------------------------------------------
09:11:56 patator    INFO - 200  194:28        0.010 | password                           |     4 | HTTP/1.1 200 O
K
```

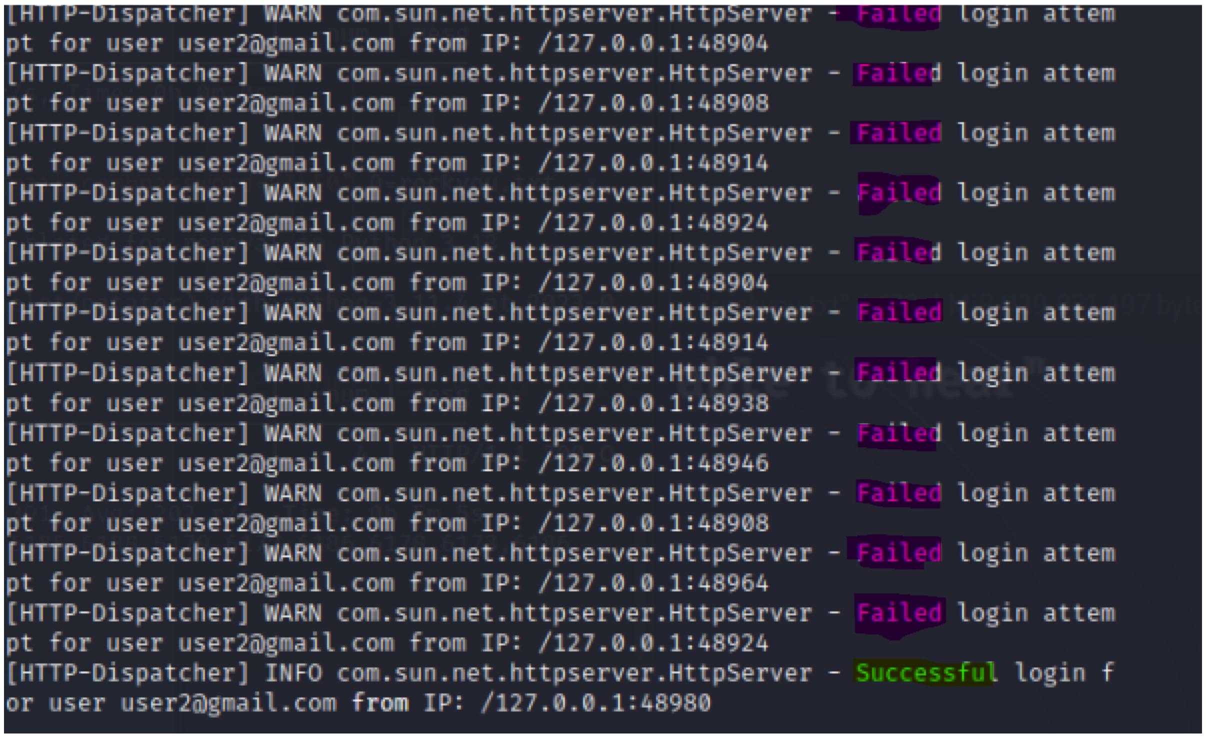
8. The command ignores PYTHONWARNINGS, calls patator to carry out a brute force attack on target url using the username set and the password from the password list provided, finally ignores code 401 which is unsuccessful login attempt
9. Patator has found the password for user2@gmail.com which was <password> in 0.01 sec

Part 1: Which security requirements is this attack violating?

This brute-force attack is violating the security requirement of Authentication and Authorization, specifically the principle that only the valid user, who knows the correct credentials (in this case, the password), should be able to log in to their account

Part 2: Show how the logging you implemented above can reveal that this vulnerability is being exploited; present a snippet from the log from the time that the exploit was performed, and explain how the snippet reveals the exploit.

Below a snippet of the logger while I was performing the attack:



```
[HTTP-Dispatcher] WARN com.sun.net.httpserver.HttpServer - Failed login attem
pt for user user2@gmail.com from IP: /127.0.0.1:48904
[HTTP-Dispatcher] WARN com.sun.net.httpserver.HttpServer - Failed login attem
pt for user user2@gmail.com from IP: /127.0.0.1:48908
[HTTP-Dispatcher] WARN com.sun.net.httpserver.HttpServer - Failed login attem
pt for user user2@gmail.com from IP: /127.0.0.1:48914
[HTTP-Dispatcher] WARN com.sun.net.httpserver.HttpServer - Failed login attem
pt for user user2@gmail.com from IP: /127.0.0.1:48924
[HTTP-Dispatcher] WARN com.sun.net.httpserver.HttpServer - Failed login attem
pt for user user2@gmail.com from IP: /127.0.0.1:48904
[HTTP-Dispatcher] WARN com.sun.net.httpserver.HttpServer - Failed login attem
pt for user user2@gmail.com from IP: /127.0.0.1:48914
[HTTP-Dispatcher] WARN com.sun.net.httpserver.HttpServer - Failed login attem
pt for user user2@gmail.com from IP: /127.0.0.1:48938
[HTTP-Dispatcher] WARN com.sun.net.httpserver.HttpServer - Failed login attem
pt for user user2@gmail.com from IP: /127.0.0.1:48946
[HTTP-Dispatcher] WARN com.sun.net.httpserver.HttpServer - Failed login attem
pt for user user2@gmail.com from IP: /127.0.0.1:48908
[HTTP-Dispatcher] WARN com.sun.net.httpserver.HttpServer - Failed login attem
pt for user user2@gmail.com from IP: /127.0.0.1:48964
[HTTP-Dispatcher] WARN com.sun.net.httpserver.HttpServer - Failed login attem
pt for user user2@gmail.com from IP: /127.0.0.1:48924
[HTTP-Dispatcher] INFO com.sun.net.httpserver.HttpServer - Successful login f
or user user2@gmail.com from IP: /127.0.0.1:48980
```

- I can clearly see from the log that there were numerous login attempts (brute-force attack) for the user user2@gmail.com and that at some point the login was Successful!

5) Problem 5: Two-Factor Authentication

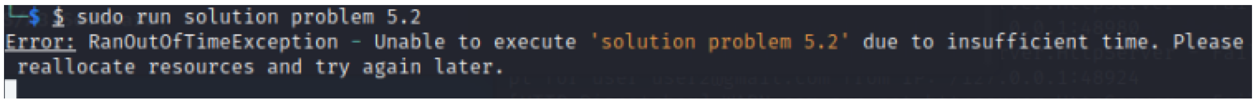
Part 1: Design a two-factor authentication protocol that uses e-mail and password. Give the protocol in protocol design syntax. Let U denote the user, C the PayBud client running in the user's browser, and S the PayBud server. Explain (in English) what each step of the protocol is.

1. U → C: User initiates login request
   User starts the login process in the client (PayBud app)
2. C → U: Request user ID and password
   Client prompts the user to enter their user ID and password

3. U → C: Insert <userID, password>
   User enters their user ID and password into the client
4. C → S: Send <userID, password>
   Client sends the user's entered credentials to the server for authentication
5. S: Check if password matches stored password for the given userID.
   If <userID, password> ∈ UserCredentials in DB
   then S→C: res with res = "One-time code required”
   else S→C: res with res = "Incorrect password"
   Server validates the password; if correct, prompts for a one-time code, else reports incorrect password
6. S: if res = "One-time code required"
   then S → E: res with res = “One-time code sent” and send with send = one-time code
   Server generates a one-time code and sends it to the user's email
7. E → U: Deliver one-time code
   User's email client delivers the one-time code to the user
8. U → C: Input one-time code
   User inputs the received one-time code into the client
9. C → S: Send one-time code
   Client sends the entered one-time code to the server for validation.
10. S: If <userID, one-time-code> is valid
    then S -> C: res with res ="Access granted" and consider user as authenticated
    else S -> C: res with res ="Access Denied"
    The server validates the one-time code; if it's correct, the user is authenticated, else access is denied
11. C: If server response is "Access granted"
    then C -> U: res with res = "Login successful"
    else C -> U: res with res = "Incorrect one-time code"
    The client informs the user of the result; either the login was successful or the one-time code was incorrect

Part 2: Modify the source code of PayBud such that it uses two-factor authentication. Explain how you modified the source code

I tried solving the second part and got the following error:



```
└─$ $ sudo run solution problem 5.2
Error: RanOutOfTimeException - Unable to execute 'solution problem 5.2' due to insufficient time. Please
 reallocate resources and try again later.
```

6) Problem 6: Attack, Man In The Middle

Part 1: PayBud is vulnerable to a man-in-the-middle attack. Why?

PayBud is vulnerable to a man-in-the-middle attack because it does not use encryption, leaving the communication between user and server exposed and manipulable

Part 2: Show the attack using protocol design syntax. Let U denote the user, C the PayBud client running in the user's browser, S the PayBud server, and A the attacker. Explain, in the run of the protocol, how the attacker obtains the login credentials. Explain what tool(s) you might use in this attack

1. U → C: User initiates login request
   User starts the login process in the client (PayBud app)
2. C → U: Request user ID and password
   Client prompts the user to enter their user ID and password
3. U → C: Insert <userID, password>
   User enters their user ID and password into the client
   3.1) A: Intercepts <userID, password>
   Attacker intercepts the communication and gets hold of the user credentials
4. C → S: Send <userID, password>
   Client sends the user's entered credentials to the server for authentication
5. S: Check if password matches stored password for the given userID
   If <userID, password> ∈ UserCredentials in DB
   then S→C: res with res = "One-time code required"
   else S→C: res with res = "Incorrect password"
   Server validates the password if correct, prompts for a one-time code, else reports incorrect password
6. S: if res = "One-time code required"
   then S → E: res with res = "One-time code sent" and send with send = one-time code
   Server generates a one-time code and sends it to the user's email
   6.1) A: Intercepts the one-time code
   Attacker intercepts the one-time code sent to the user's email
7. E → U: Deliver one-time code
   User's email client delivers the one-time code to the user
8. U → C: Input one-time code
   User inputs the received one-time code into the client
   8.1) A: Intercepts and changes one-time code
   Attacker intercepts the one-time code and alters it before it is sent to the server
9. C → S: Send altered one-time code Client sends the altered one-time code to the server for validation.
10. S: If <userID, altered one-time-code> is valid
    then S -> C: res with res ="Access granted" and consider user as authenticated
    else S -> C: res with res ="Access Denied"
    The server validates the one-time code; if it's correct, the user is authenticated, else access is denied

11. C: If server response is "Access granted"
    then C -> U: res with res = "Login successful"
    else C -> U: res with res = "Incorrect one-time code"
    The client informs the user of the result; either the login was successful or the one-time code was incorrect

Part 3: Can logging reveal such an exploit? How/Why-not?

Yes, logging can potentially reveal such an exploit. Detailed server-side logs, like those that track IP addresses and session details, could highlight suspicious activity such as multiple login attempts from different locations for the same user account or unexpected alterations in the sequence of events, raising a flag for possible man-in-the-middle attacks