SIOProject_1

Repository for SIO 2023/2024 1st Project

Project Members

- **Diogo Marto - 108298**
- **Tiago Pereira - 108546**

Project Description

This project theme is "DETI memorabilia at the University of Aveiro". The aim is to develop a functional webstore with concealed vulnerabilities that are not apparent to casual users but can be exploited to compromise the system. As required, we present both a flawed and a corrected version of the shop, detailing how these vulnerabilities are explored and their impact.

Explored CWE's

BaseImproper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

Weak Authentication

Direct Request ('Forced Browsing')

Improper Neutralization of Special Elements Used in a Template Engine

Installation

You need to have docker installed and the docker daemon running. Inside the folders app or app_sec run the following command to create a docker image:

```
$ docker build -t [name_of_image] .
```

Note you need to make this for both app and app_sec with differnt names for the image.

Execution

You need to have docker installed and the docker daemon running. Run the following command

```
$ docker run -p [desired_port_outside]:5000 [name_of_image]
```

Note make sure desired port is not being used and name of image is the ones created on Execution section.

License

MIT

Analysis CWE-89: SQL Injection

Introduction

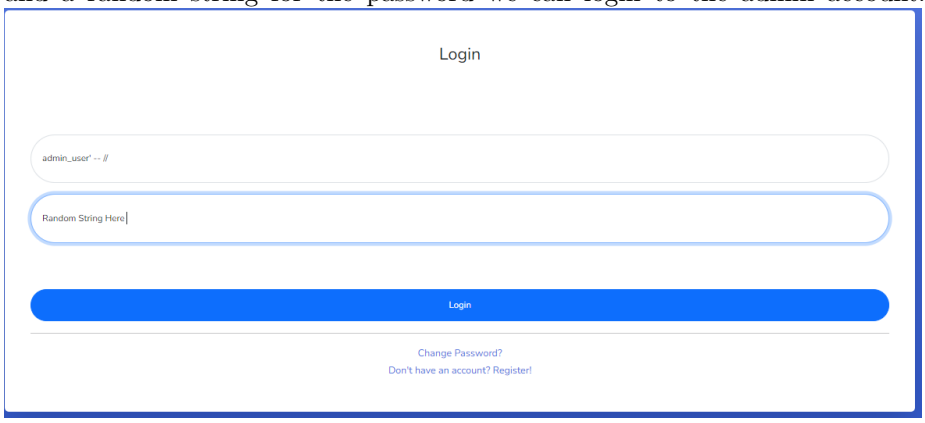
SQL injection attacks represent a serious threat to any database-driven site.The methods behind an attack are easy to learn and the damage caused can range from considerable to complete system compromise.

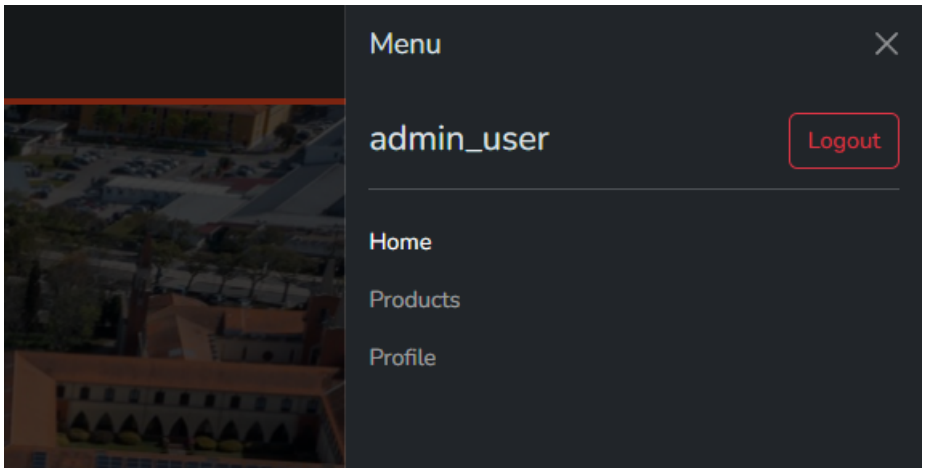
Vulnerabilities

Basic Injection to Bypass Login

Example

We have a simple login form on http://127.0.0.1:5000/auth/login and we also have an admin user called `admin_user`. But this form is vulnerable to a simple sql injection. By simply writing `admin_user'-- //` on the username and a random string for the password we can login to the admin account.





Weak code

```
user = db.execute(
    "SELECT * FROM user WHERE Username ='" + username + "' AND Password='" + generate_password_hash(password) + "'"
).fetchone()

if user is None:
    error = 'Incorrect credentials.'
```

Figure 1: image

We are making the sql command based on string concatenation which allows to do a simple sql injection. (Blueprints/auth.py login() function)

Fix

- We can use a builtin functionality from the sqlite3 python library that allows us to build safe sql commands very simply.

```
user = db.execute(
    "SELECT * FROM user WHERE Username=? AND Password=?",(username,generate_password_hash(password))
).fetchone()

if user is None:
    error = 'Incorrect credentials.'
```

Figure 2: image

- We can refactor the logic of our login so that it isnt vulnerable to this attack. In this example the `check_password_hash()` must always run and cant be skipped.
- We could also make an udf that would achieve the same thing that our functionality from sqlite3 , separation between arguments and the sql command.

```python
user = db.execute(
    'SELECT * FROM user WHERE Username = ?', (username,)
).fetchone()

if user is None:
    error = 'Incorrect username.'
elif not check_password_hash(user['Password'], password):
    error = 'Incorrect password.'
```

Figure 3: image

Basic injection on product listing to get names of all users

Example

We can search for products using a string but this field is vulnerable to sql injection.

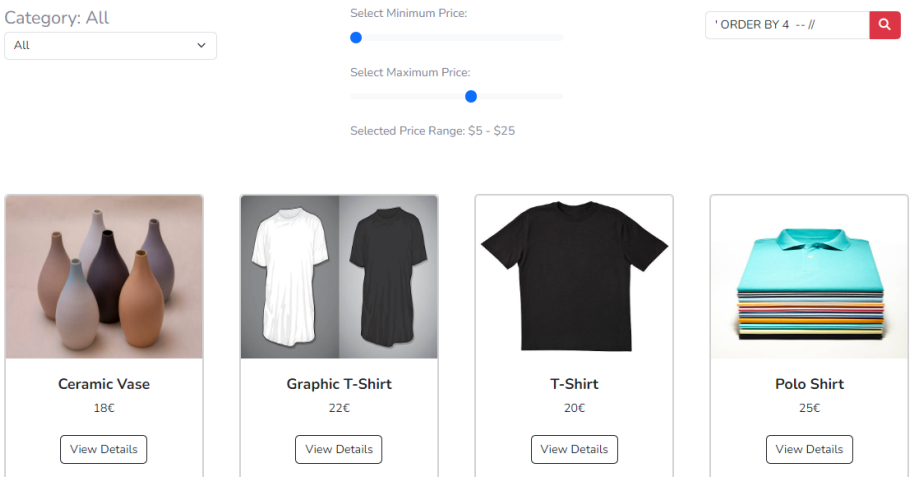


Figure 4: image

`' ORDER BY 4 -- //` We can check that is vulnerable by ordering the products in a different order also we can see that we can only order up to the 4th collumn. After we know that we are working with 4 collumns we can add information to the product listing. For example with `' UNION SELECT 1,2,3,4,5 -- //`

But we can cause a bit more damage with `' UNION SELECT  Username , ID , Password , Role FROM User ORDER BY 2 -- //`

Which gives us a list of the username and the corresponding encrypted password.

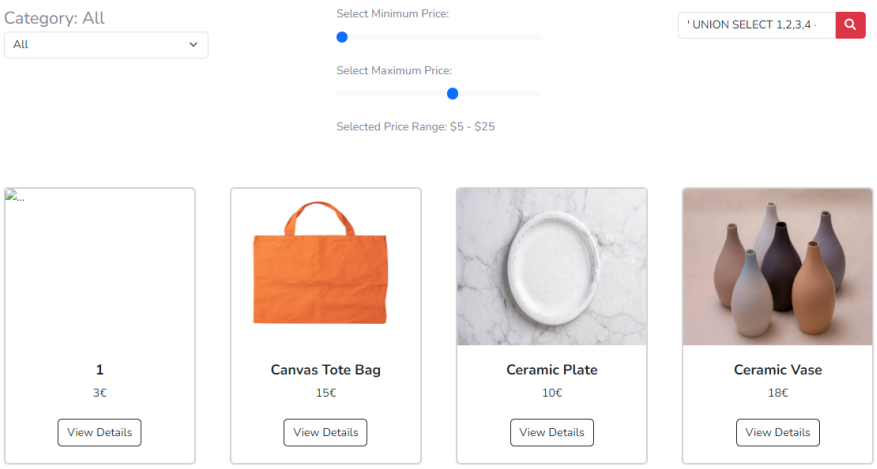


Figure 5: image

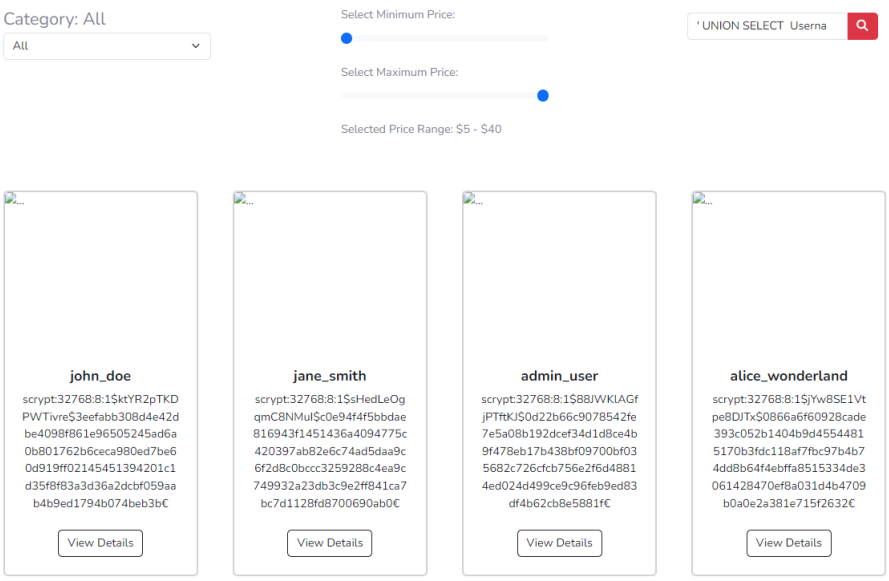


Figure 6: image

Weak code

```
if category is None or len(category)==0 or category=="All" or category=="None":
    items=db.execute("SELECT * FROM Product Where Price >=" + str(minPrice) +
                     " AND Price <="+str(maxPrice)+
                     " AND Product.Name LIKE '"+search+"'").fetchall()
    category="All"
else:
    items=db.execute("SELECT * FROM Product JOIN Category_Has_Product ON Cname='"+category+"' WHERE Pname=Product.Name"+
                     " AND Price >=" + str(minPrice) +
                     " AND Price <=" + str(maxPrice)+
                     " AND Product.Name LIKE '"+search+"'").fetchall()
```

Figure 7: image

Once again we are building our SQL commands using concatenation of strings. (Blueprints/shop.py products() function)

Fix

- We can deploy a similar strategy used above and use the sqlite3 python functionality to prevent SQL injection.

```
if category is None or len(category)==0 or category=="All" or category=="None":
    items=db.execute("SELECT * FROM Product Where Price >= ? AND Price <= ? and Product.Name LIKE ? ",(minPrice,maxPrice,search,)).fetchall()
    category="All"
else:
    items=db.execute("SELECT * FROM Product JOIN Category_Has_Product ON Cname=? WHERE Pname=Product.Name and Price >= ? AND Price <= ? and Product.Name LIKE ?",
                     (category,minPrice,maxPrice,search,)).fetchall()
```

Figure 8: image

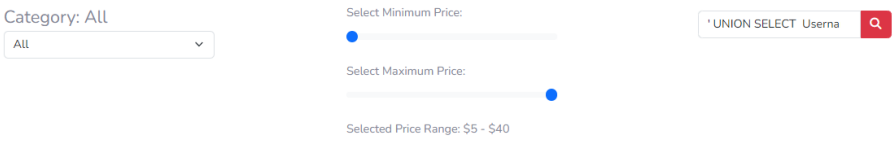


Figure 9: image

SQL Second order attacks

Example

When register our Username can be anything due to that we can store an unsanitized piece of SQL. Like `admin_user' -- //`

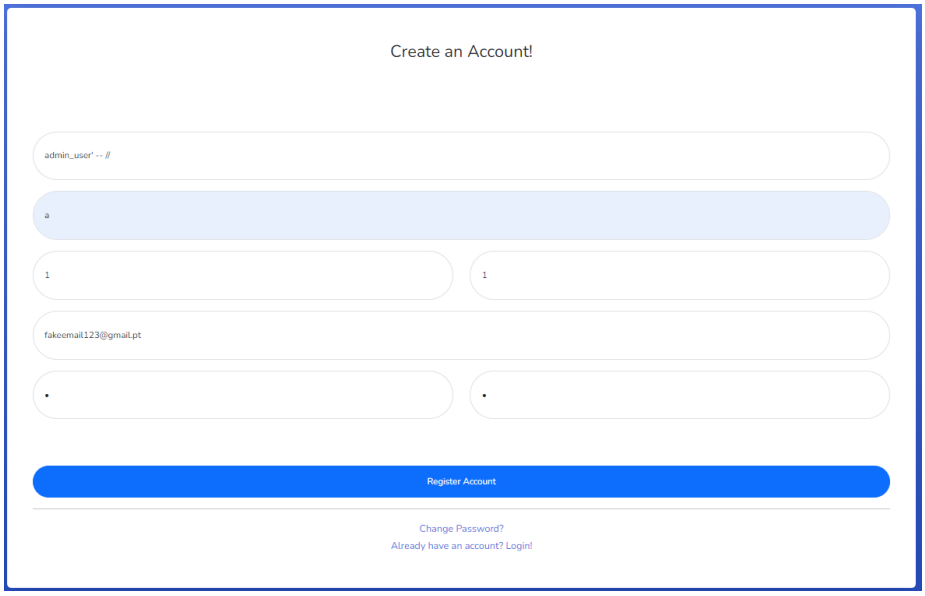


Figure 10: image

Now we can login into this new account , but funnily enough since our login is vulnerable to SQL injections we cant even login to this account since instead it will login to the `admin_user`. So we have a route `auth/login/safe` that doesnt have SQL injection we can login to `admin_user' -- //`

Now if we go to the profile we will see something interesting.

We see the profile of the `admin_user` !

Weak code

On Blueprints/shop.py on the function profile() we have this segment of code. On `g.user` its stored our current session that as all the some details about the user namely `Username`. When we store an unsanitized piece of SQL in `Username` the resulting SQL command built from concatenation will be unsafe in our case will give us the profile for the `admin_user`.

Fix

- We can once again use the builtin functionality from the sqlite3 python library that allows us to build safe sql commands and even if the `Username` is unsafe it still be treated in a manner thats safe. In the same vein we can

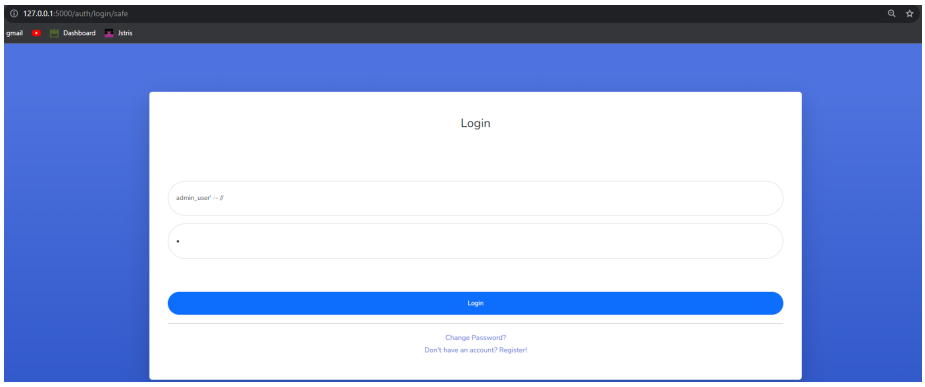


Figure 11: image

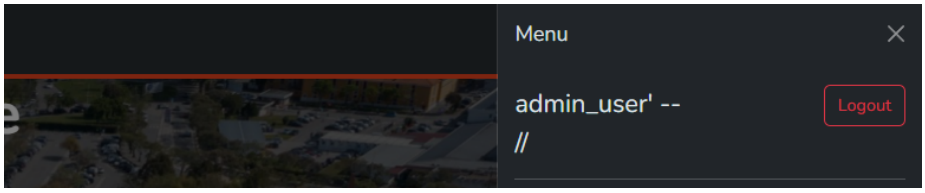


Figure 12: image

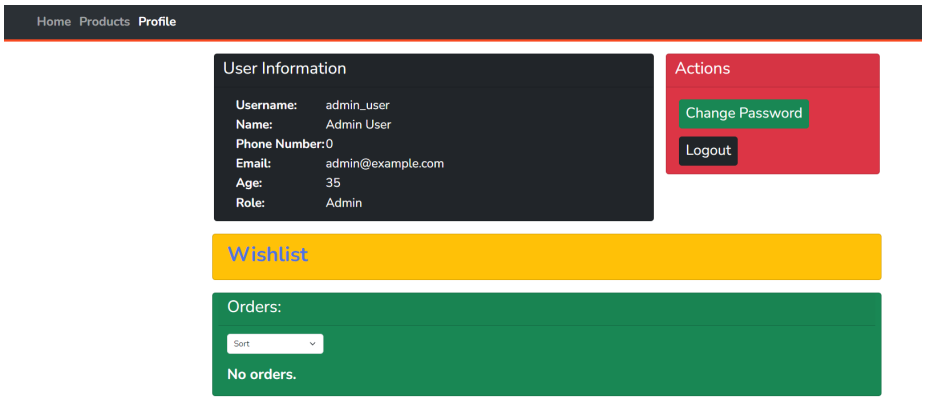


Figure 13: image

```
user = db.execute("SELECT Username,Name,PhoneNumber,Email,Age,Role FROM User Where Username = '"+g.user["Username"]+"'").fetchone()
orders = db.execute("SELECT * FROM [Order] Where Client = '"+g.user["Username"]+"'").fetchall()
```

Figure 14: image

build the SQL based on something the user doesnt have control namely `ID` which is generated by the database.

```
user = db.execute("SELECT Username,Name,PhoneNumber,Email,Age,Role FROM User Where ID = ?",(g.user["ID"],)).fetchone()
orders = db.execute("SELECT * FROM [Order] Where Client = ?",(user["Username"],)).fetchall()
```

Figure 15: image

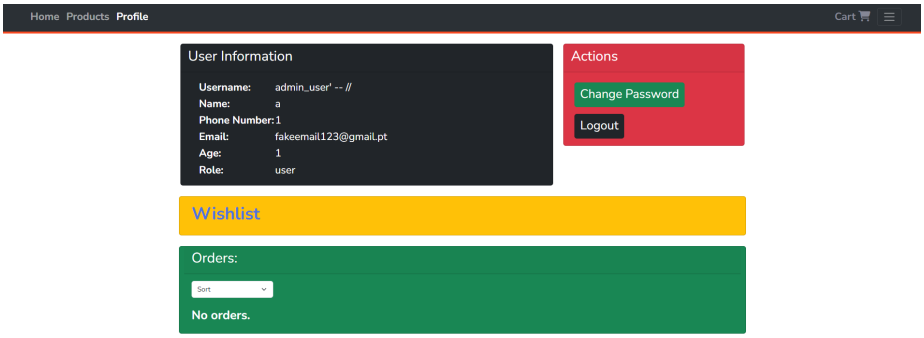


Figure 16: image

- Another fix is proper validation of what we store, in our case what we store on `Username`. We can assume that `Username` can only be alphanumeric for example.

```
if not username.isalnum():
    error += ['Username is not alphanumeric.\n']
```

Figure 17: image

On Blueprints/auth.py register() function

Blind Injection

Example

We can search for products using a string but this field is vulnerable to sql injection.

We can see that products will only appear when on `' AND TEST -- //` the `TEST` is true. For example we can use `' AND ( select COUNT(*) from User ) > NUM -- //` to see if the number of users is superior to `NUM` And by doing some queries we can find exact amount. `- ' AND ( select COUNT(*) from User ) > 50 -- //` is False. `- ' AND ( select COUNT(*) from User ) > 25 -- //` is False. `- ' AND ( select COUNT(*) from User ) > 12 -- //` is True. `- ' AND ( select COUNT(*) from User ) > 18 -- //` is True.

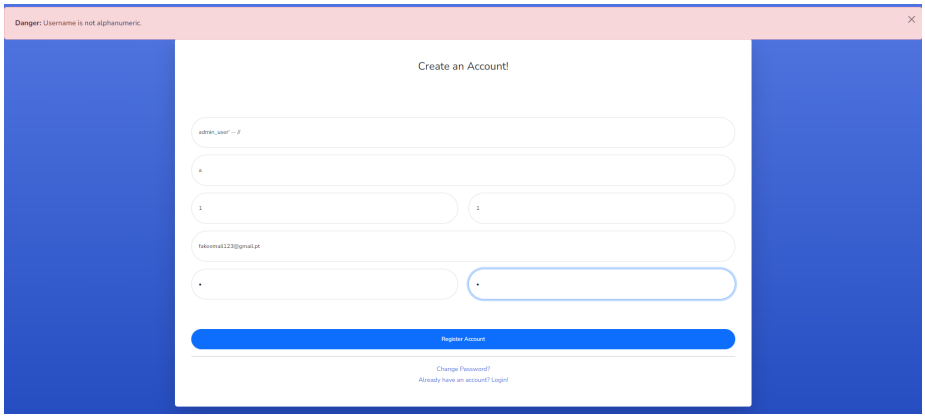


Figure 18: image

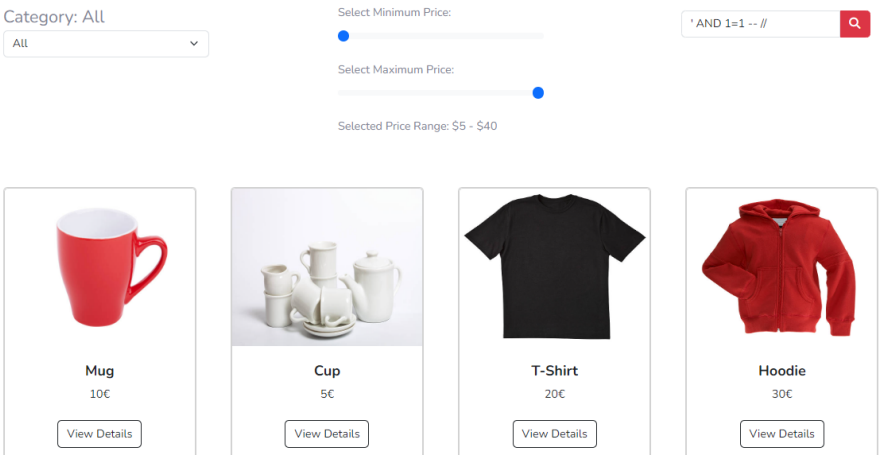


Figure 19: image

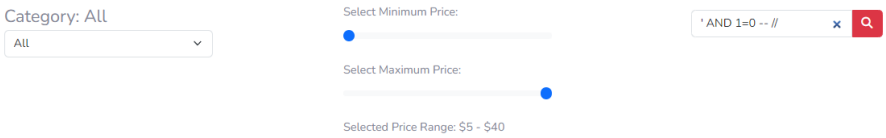


Figure 20: image

- `' AND ( select COUNT(*) from User ) > 21  -- //` is False. - `' AND ( select COUNT(*) from User ) > 20  -- //` is True. So we get that we have 21 users.

Weak code

```
if category is None or len(category)==0 or category=="All" or category=="None":
    items=db.execute("SELECT * FROM Product Where Price >=" + str(minPrice) +
                     " AND Price <="+str(maxPrice)+
                     " AND Product.Name LIKE '"+search+"'").fetchall()
    category="All"
else:
    items=db.execute("SELECT * FROM Product JOIN Category_Has_Product ON Cname='"+category+"' WHERE Pname=Product.Name"+
                     " AND Price >=" + str(minPrice) +
                     " AND Price <=" + str(maxPrice)+
                     " AND Product.Name LIKE '"+search+"'").fetchall()
```

Figure 21: image

Once again we are building our SQL commands using concatenation of strings. (Blueprints/shop.py products() function)

Fix

- We can deploy a similar strategy used above and use the sqlite3 python functionality to prevent SQL injection.

```
if category is None or len(category)==0 or category=="All" or category=="None":
    items=db.execute("SELECT * FROM Product Where Price >= ? AND Price <= ? and Product.Name LIKE ? ",(minPrice,maxPrice,search,)).fetchall()
    category="All"
else:
    items=db.execute("SELECT * FROM Product JOIN Category_Has_Product ON Cname=? WHERE Pname=Product.Name and Price >= ? AND Price <= ? and Product.Name LIKE ?",
                     (category,minPrice,maxPrice,search,)).fetchall()
```

Figure 22: image

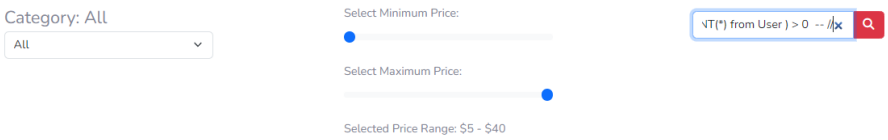


Figure 23: image

Analysis CWE-79: XSS

Introduction

XSS attacks exploit vulnerabilities within Web interactions where an attacker performs indirect actions against Web clients through a vulnerable Web application. The primary result is that some external code is injected into the victim's web browser and will be executed. All existing context, including valid cookies, as well as computational resources of the victim become available to the attacker.

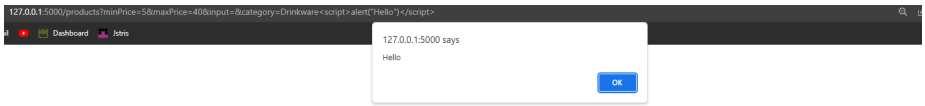
Vulnerabilities

Reflected XSS Attack

Example

We can search for products based on a url , i.e http://127.0.0.1:5000/products?minPrice=5&maxPrice=40&input=
. But we can exploit this to do a reflected XSS attack , the field of category is rendered on the page by changing it we can inject unwanted code in the website.

This link, http://127.0.0.1:5000/products?minPrice=5&maxPrice=40&input=&category=Drinkware%3Cscript%
is compromised when the page loads it shows and a popup saying "Hello".



Weak code

```
<span class="fs-4">Category: {{category | safe }}</span>
```

Figure 24: image

In this piece of code we assume that category is safe so it can be treated has html. (templates/products.html)

Also on the function products() on Blueprints/shop.py we never validate if the category field is an existing one or if it just garbage.

Fix

- If we omit the safe tag jinga2 (our template engine) will assume that the input is unsafe and only treat it has text.

```
<span class="fs-4">Category: {{ category }}</span>
```

Figure 25: image

- If we validate if the category exists we can on the server indicate correct steps to fix the problem.

Stored XSS Attack

Example

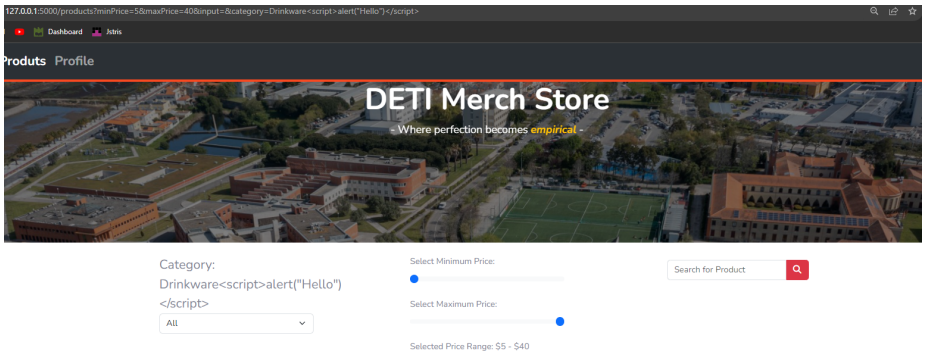


Figure 26: image

```
existCategory = db.execute("SELECT * FROM Category WHERE Name = ?", (category,)).fetchone()
print(existCategory)
if not existCategory and category!="All":
    error += ["Category doesnt exist"]
    category="All"
```

Figure 27: image

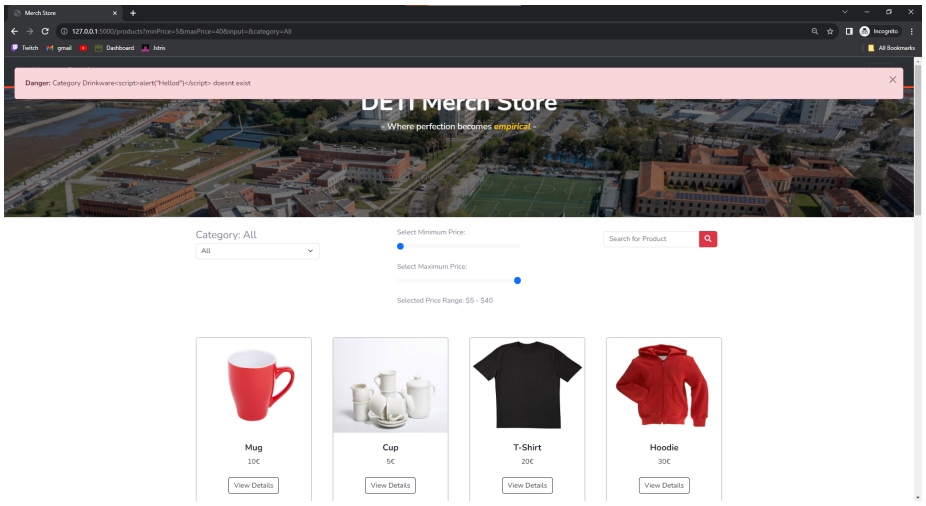


Figure 28: image

We can write reviews for a product.

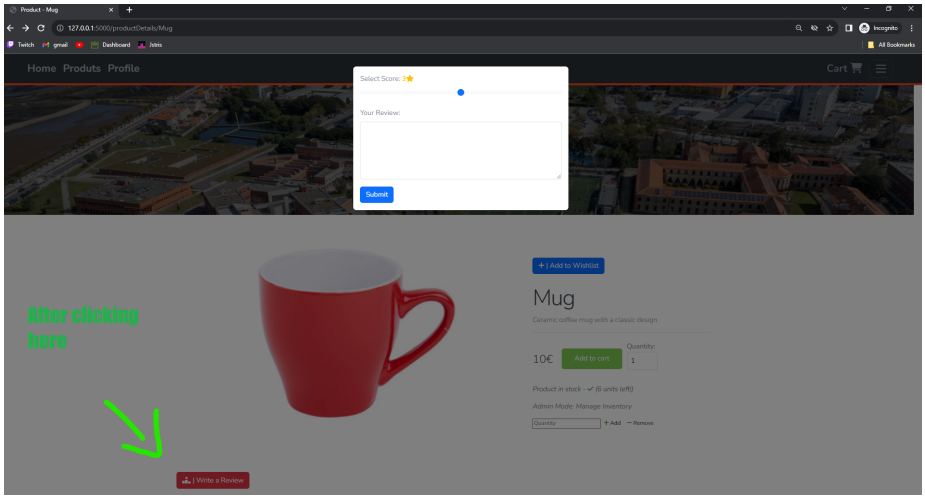


Figure 29: image

But we can insert a malicious payload in the review that will then be stored on the server. Users then accessing this webpage will render the malicious payload.

After we load the page again we get:

Weak code

We assume that the review is safe so the contents of it are treated like html.

Also we dont check on the server side if the review migth have malicious code.

Fix

- Just omit the safe tag and the review will only be treated as text.

```
<p class="card-text">
    {{ review.ReviewBody }}
</p>
```

- We could also parse the review and check if it has any potencial malicous payload and reject it but for our case we assumed that the review is only text so this is uneccesary and just removing the safe tag is enough.

CSRF Attack

Example

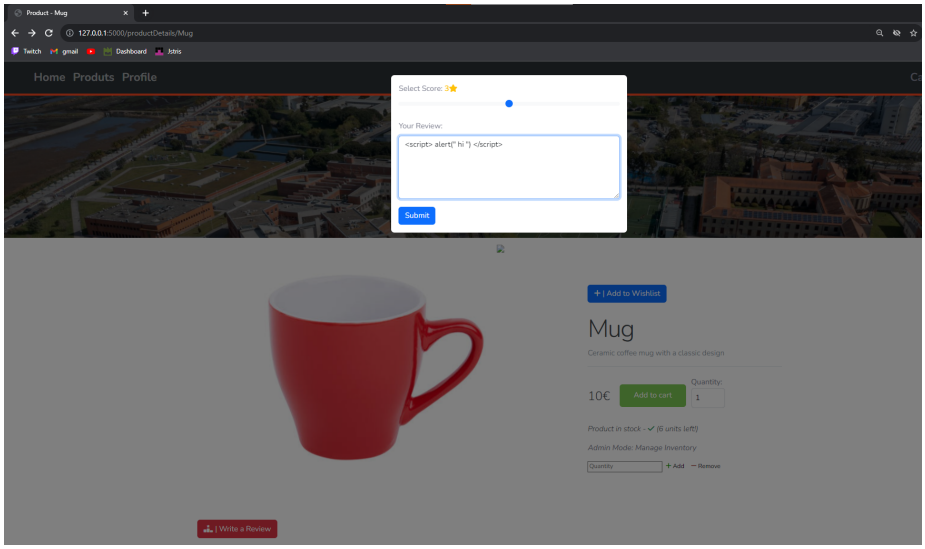


Figure 30: image

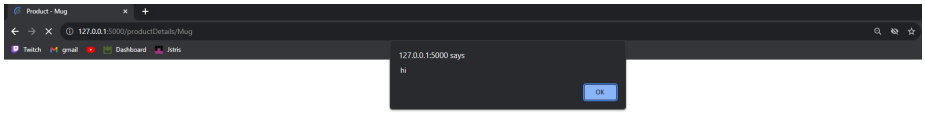


Figure 31: image

```
<p class="card-text">
    {{ review.ReviewBody | safe }}
</p>
```

Figure 32: image

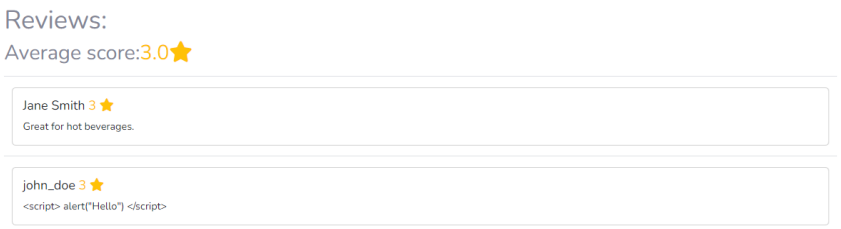


Figure 33: image

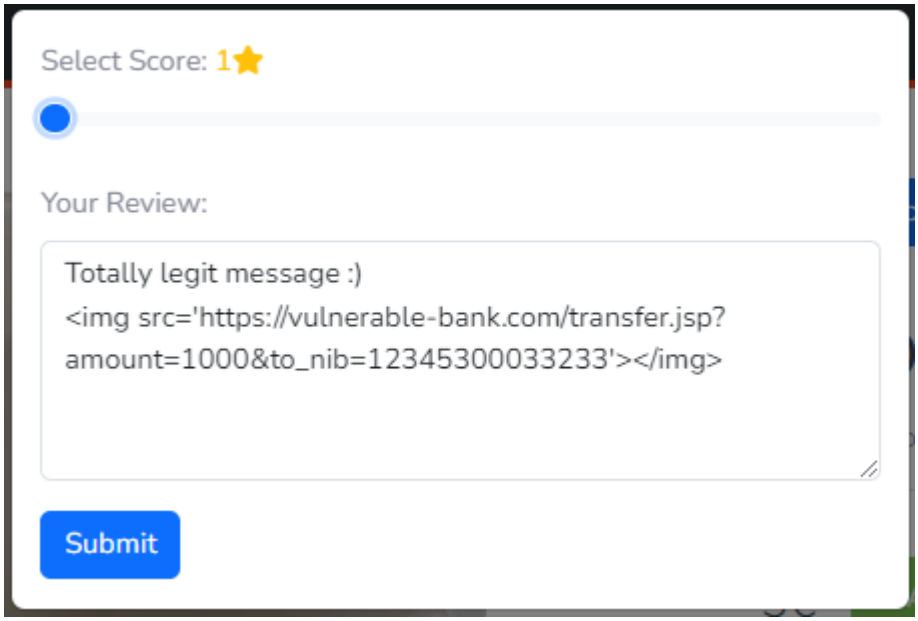


Figure 34: image

We can also target different servers using an XSS attack in this example we use the image tag to make a request to a server with the credentials of a user who visits the page with the malicious review.

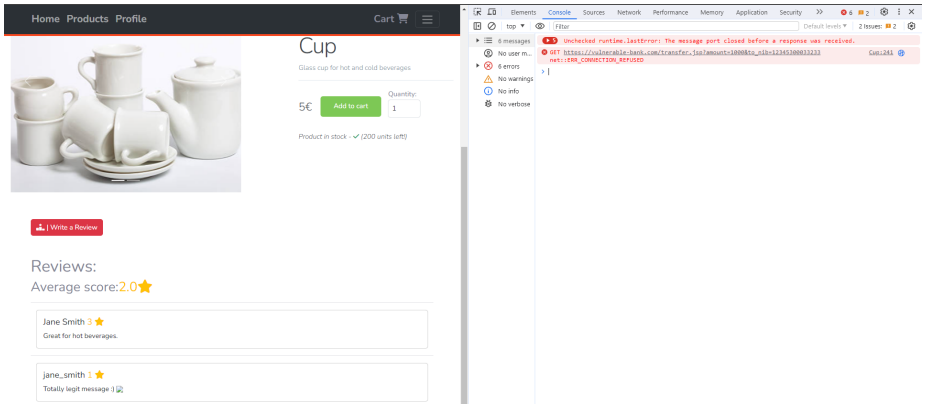


Figure 35: image

As we can see a GET request to `https://vulnerable-bank.com/transfer.jsp?amount=1000&to_nib=123453` was sent and a normal user migth have been unware of this.

Fix

- Don´t allow stored XSS Attack and reflected XSS Attack with the strategies diccussed above.
- Enable CORS or a secure Content Security Policy so that the browser knows what to not load.

```
Refused to load the image 'https://vulnerable-bank.com/transfer.jsp?amount=1000&to_nib=1234530 Cup:255
0033233' because it violates the following Content Security Policy directive: "img-src 'self' https://m
edia.gettyimages.com https://pbs.twimg.com data:".
```

Figure 36: image

Analysis CWE-1390: Weak Authentication

Introduction

It's common for users to prefer the definition and use of weak password, i.e., sequences, common words, etc... Why is that? It's pratical and easy to remember. But that provides a perfect environment for credentials theft, as it is easy to break those credentials with a simple approach... Brute force.

Vulnerability code:

In auth.py, when we handle the definition and change of a password, no verification to its complexity is being made, allowing for the insertion of weak passwords.

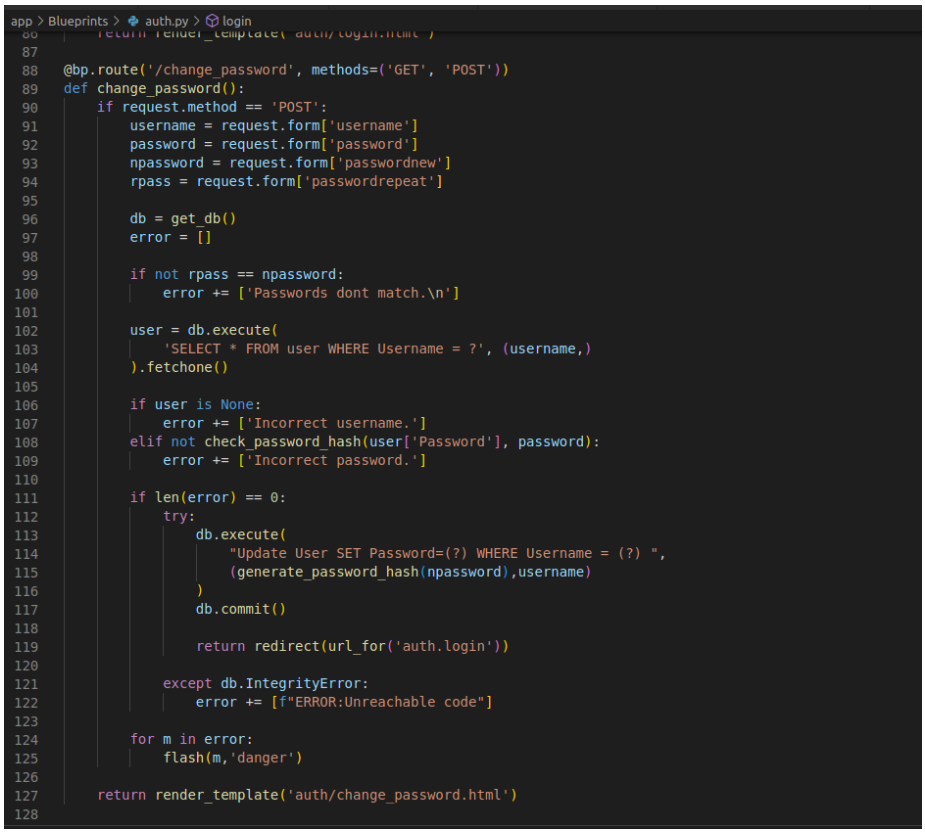


Figure 37: images

Exploit

Consider the following simple script in python, present in the bruteforce.py file:

```python
import requests

link="http://localhost:5000/auth/login/safe"

file1 = open('10-million-password-list-top-10000.txt', 'r')
Lines = file1.readlines()
count=0
for line in Lines:
    r=requests.post(link,data={"username" : "Weak", "password" : line.strip()})
    if r.url!=link:
        print("Try ",count,": Password found! -> '", line.split(),"'")
        break
    count+=1
```

Note: We are using a github document, stored in "10-million-password-list-top-10000.txt"(obtained from: https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/10-million-password-list-top-1000000.txt)

We have a user called Weak. He uses a very basic password. Through brute force, fetching from a common password database we we're able to find it's password. It was "potato":

```
Try 2087: Password found! -> 'potato'
(venv) tiago@Tigas:~/UA/3 Ano/1 Semestre/SIO/Proj1/SIOProject_1/analysis$
```

Figure 38: images

Solution?

To solve this problem we can take a very simple approach. Enforce strong passwords through password validation. To achieve this, we created a function that checks if the password is acceptable according to the following requirements: **-> have a mix of uppercase and lowercase letters -> have at least one digit -> have at least one special character -> have a minimum length of 8 characters**

Secure Code:

We validate:

```python
def password_strong(password):
    pattern = r'^(?=.*[A-Z])(?=.*[a-z])(?=.*\d)(?=.*[@#$%^&!])[A-Za-z\d@#$%^&!]{8,}$'
    return re.match(pattern, password) is not None
```

And act accordingly:

```
    if not email:
        error += ['Email is required.\n']
    if not rpass == password:
        error += ['Passwords dont match.\n']

    if len(error) == 0:
        if not password_strong(password):
            error+=[("Password must be a mix of uppercase and lowercase letters, have at least one digit, at l
        else:
            try:
                db.execute(
                    "INSERT INTO User (Username, Password, Name , PhoneNumber , Email , Age , Role) VALUES (?,
                    (username, generate_password_hash(password) , name , phone , email , age , 'user')
                )
                db.commit()
            except db.IntegrityError:
                error += [f"User {username} is already registered."]
            else:
                return redirect(url_for("auth.login"))

    for m in error:
        flash(m,'danger')

return render_template('auth/register.html',title="Register")
```

Analysis CWE-425:Forced Browsing

Introduction

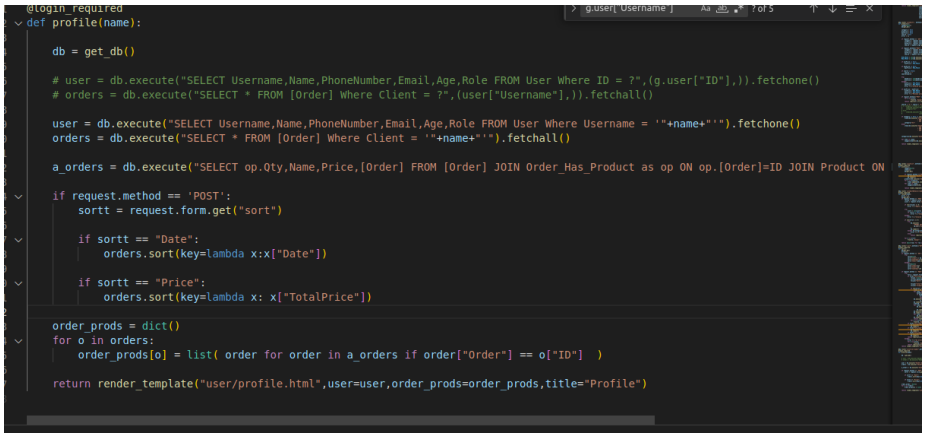
Imagine being able to access classified information without the necessary credentials with the excuse of "I want to!". As naive as it sounds, thats precisely what Forced Browsing stands for. It envolves manipulating front-end elements to access whats not supposed to be accessed.

Vulnerability code:

In shop.py when we handle the profile route, we consider the username as input parameter passed through the URL. That allows the external world to manipulate and forge a fake identity, accessing other users sensible information(such as billing).

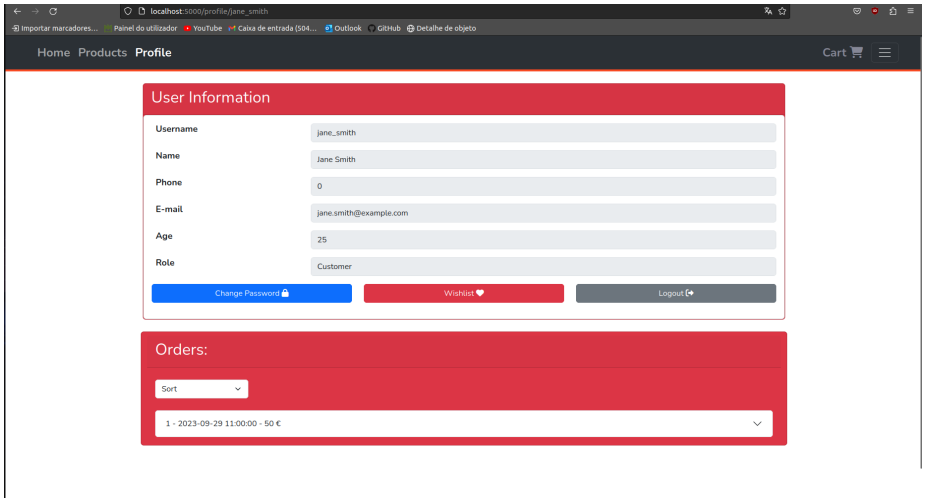
Exploit

We can manage the URL to access the information about whoever we want to. Username information can be obtained through other methods, such as fishing and pool of common usernames. Some examples of this exploit:



```
@login_required
def profile(name):

    db = get_db()

    # user = db.execute("SELECT Username,Name,PhoneNumber,Email,Age,Role FROM User Where ID = ?",(g.user["ID"],)).fetchone()
    # orders = db.execute("SELECT * FROM [Order] Where Client = ?",(user["Username"],)).fetchall()

    user = db.execute("SELECT Username,Name,PhoneNumber,Email,Age,Role FROM User Where Username = '"+name+"'").fetchone()
    orders = db.execute("SELECT * FROM [Order] Where Client = '"+name+"'").fetchall()

    a_orders = db.execute("SELECT op.Qty,Name,Price,[Order] FROM [Order] JOIN Order_Has_Product as op ON op.[Order]=ID JOIN Product ON

    if request.method == 'POST':
        sortt = request.form.get("sort")

        if sortt == "Date":
            orders.sort(key=lambda x:x["Date"])

        if sortt == "Price":
            orders.sort(key=lambda x: x["TotalPrice"])

    order_prods = dict()
    for o in orders:
        order_prods[o] = list( order for order in a_orders if order["Order"] == o["ID"]  )

    return render_template("user/profile.html",user=user,order_prods=order_prods,title="Profile")
```

Figure 39: images



We have done it all without the need of any type of authentication!

Solution?

To solve this problem we can take a very simple approach. Pass the username variable, which is constant through a session as as internalized variable, not allowing front-end manipulation of sensible data accessors.

Secure Code:

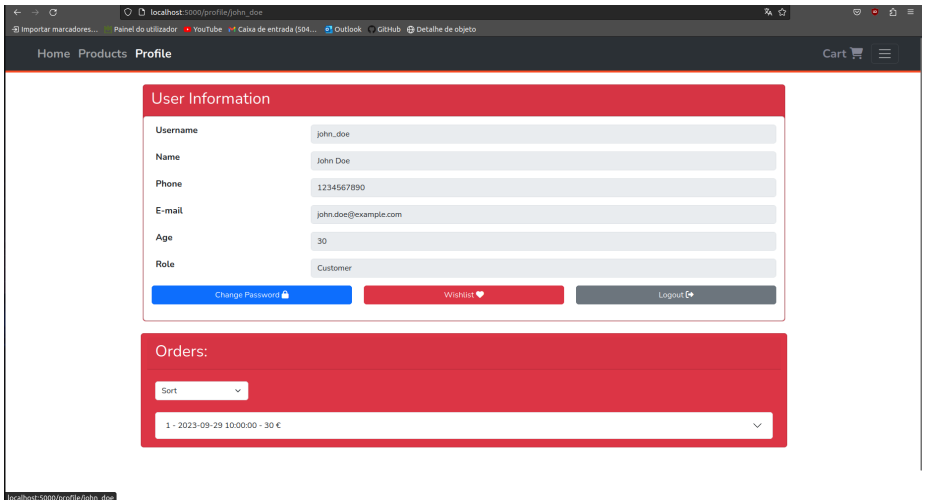
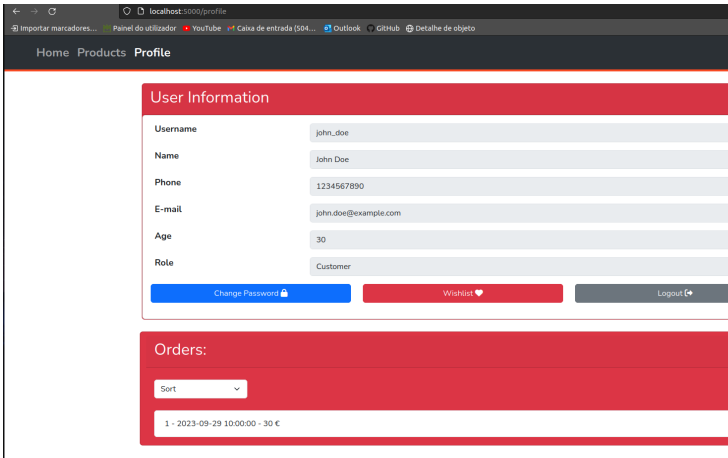


Figure 40: images

We use the internal variable g.user[“Username”]:

```
@bp.route("/profile",methods=("GET","POST"))
@login_required
def profile():

    db = get_db()

    # user = db.execute("SELECT Username,Name,PhoneNumber,Email,Age,Role FROM User Where ID = ?",(g.user["ID"],)).fetchone()
    # orders = db.execute("SELECT * FROM [Order] Where Client = ?",(user["Username"],)).fetchall()

    user = db.execute("SELECT Username,Name,PhoneNumber,Email,Age,Role FROM User Where Username = '"+g.user["Username"]+"'").fetchone()
    orders = db.execute("SELECT * FROM [Order] Where Client = '"+g.user["Username"]+"'").fetchall()

    a_orders = db.execute("SELECT op.Qty,Name,Price,[Order] FROM [Order] JOIN Order_Has_Product as op ON op.[Order]=ID JOIN Product ON PName=NAME Where

    if request.method == 'POST':
        sortt = request.form.get("sort")

        if sortt == "Date":
            orders.sort(key=lambda x:x["Date"])

        if sortt == "Price":
            orders.sort(key=lambda x: x["TotalPrice"])

    order_prods = dict()
    for o in orders:
        order_prods[o] = list( order for order in a_orders if order["Order"] == o["ID"]  )

    return render_template("user/profile.html",user=user,order_prods=order_prods,title="Profile")
```

And the URL has no longer the name parameter:



Analysis CWE-1336: Improper Neutralization of Special Elements Used in a Template Engine

Introduction

Many web applications use template engines that allow developers to insert externally-influenced values into free text or messages in order to generate a full web page, document, message, etc. Such engines include Twig, Jinja2, Pug, Java Server Pages, FreeMarker, Velocity, ColdFusion, Smarty, and many others - including PHP itself. Some CMS (Content Management Systems) also use templates.

Template engines often have their own custom command or expression language. If an attacker can influence input into a template before it is processed, then the attacker can invoke arbitrary expressions, i.e. perform injection attacks. For example, in some template languages, an attacker could inject the expression "{{7*7}}" and determine if the output returns “49” instead. The syntax varies depending on the language.

We our case we are using Jinja2 to generate this templates.

Example

We can search for products based on a url , i.e `http://127.0.0.1:5000/products?minPrice=5&maxPrice=40&i` . But we can exploit this since our category field is vulnerable to SSTi. The link `http://127.0.0.1:5000/products?minPrice=5&maxPrice=40&input=&category={{2*2}}` returns the following result.

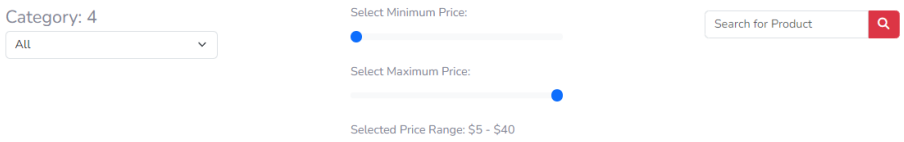


Figure 41: image

As we can see the category is 4 the result of 2*2. But this raises the question who did this calculation and why? The answer is that the server did , so that means that we are executing code on the server and the server ran the code for the calculation because `{{ }}` signals to Jinga2 to render something that it in our case the result of 2*2. If we try the link `http://127.0.0.1:5000/products?minPrice=5&maxPrice=40&input=&category={{print("Hi")}}` we will see something curious.

This error is the result of the template engine running in a sandbox where we

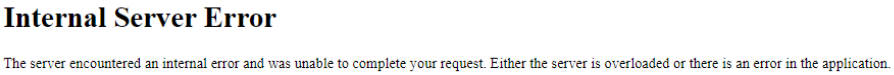


Figure 42: image

cant do function calls directly i.e we cant do `print("Hi")`. So what do we have access to? We at least have access to numbers as we show above , but also strings , list , dict , tuple , some variables like config , request .

`http://127.0.0.1:5000/products?minPrice=5&maxPrice=40&input=&category={{%20config%20}}` returns the following result.

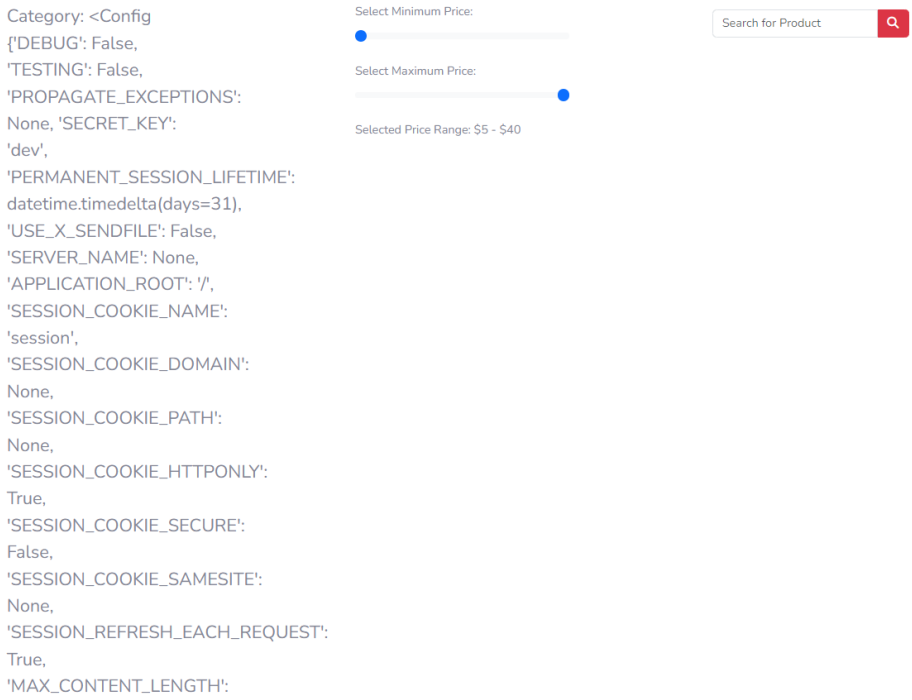


Figure 43: image

In here we have access to something we probably shouldnt have, namely 'SECRET_KEY': 'dev'.

`http://127.0.0.1:5000/products?minPrice=5&maxPrice=40&input=&category={{%20dict.__base__.__s` returns the following result.

What happened here is that we are running this piece of code `dict.__base__.__subclasses__()`. `dict` is a class and `dict.__base__` returns the base class of `dict` which is

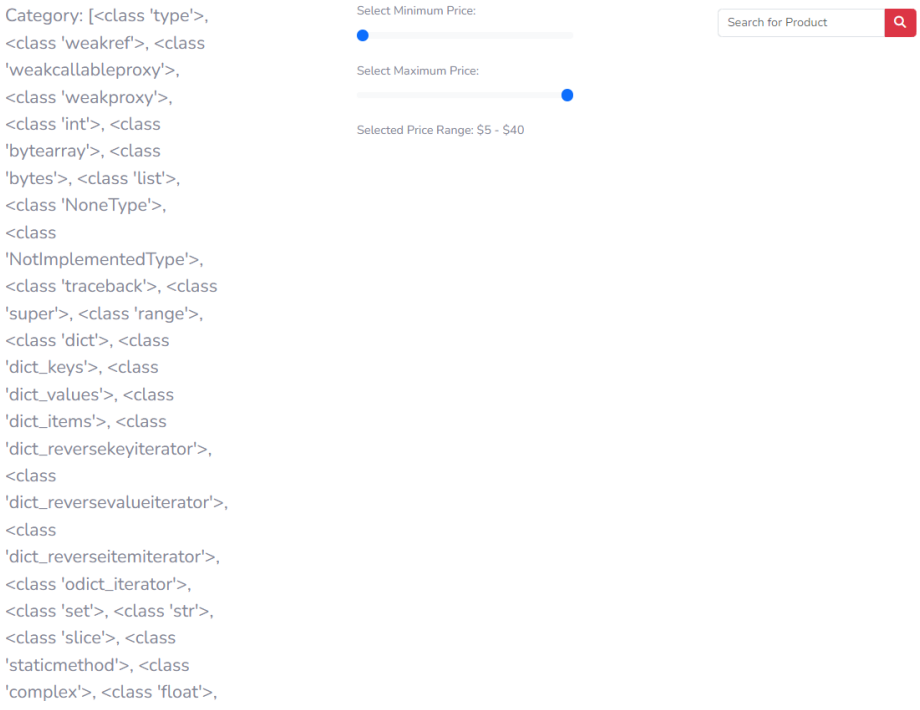


Figure 44: image

`Object`. Then we basicly do `Object.subclasses()` which returns a list of all classes (since all classes come from the `Object` class). And this has give us the opportunity to access hundreds of new classes and functions in those classes , and have some fun with them. `{{dict.__base__.__subclasses__()[351]}}` is gonna give us `<class 'subprocess.Popen'>` which spawn some process. (the index to find this class may vary)

`http://127.0.0.1:5000/products?minPrice=5&maxPrice=40&input=&category={{dict.__base__.__subc`

Lists the file on the directory with `ls`.

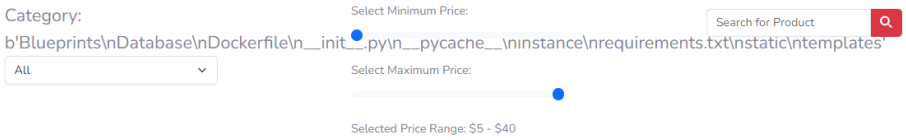


Figure 45: image

`http://127.0.0.1:5000/products?minPrice=5&maxPrice=40&input=&category={{dict.__base__.__subc`

Shows the contents of `__init__.py` which is source code of our website.

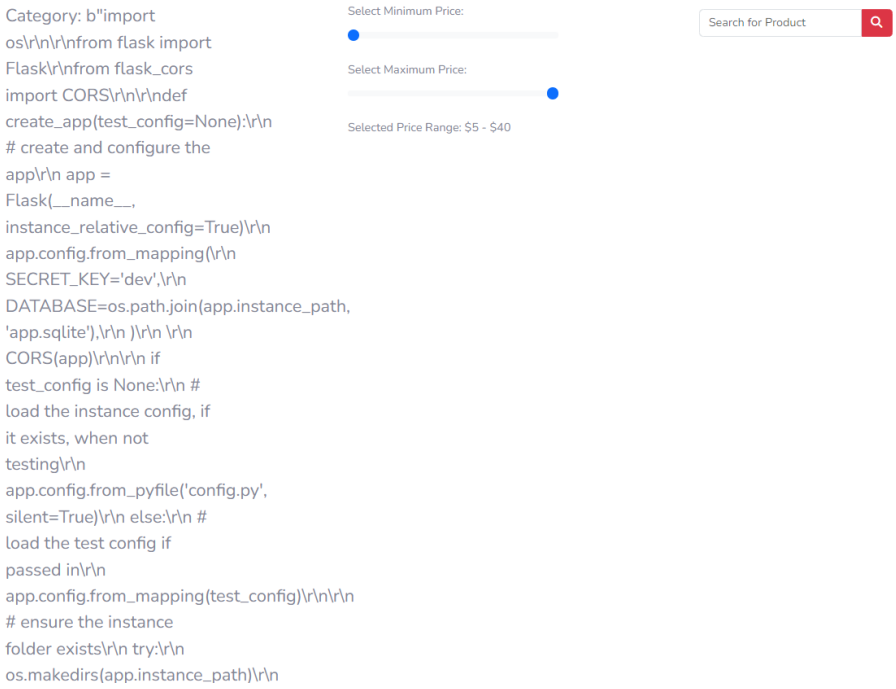


Figure 46: image

At this point we have access to `bash` commands and we can go some damage to the server.

Weak code

```
category=render_template_string(category)
```

Figure 47: image

On this line we are calling rendering on category as a template but we never check if category is safe to render().

Fix

- Category shouldnt be rendered as a template it should be only treated as text to inject in some template, never a template.

```
render_template('product/products.html',
                search=search[1:-1],
                categories=categories,
                products=items,
                title="Merch Store",
                images=images,
                category=category,
                MIN_PRICE=MIN_PRICE,
                MAX_PRICE=MAX_PRICE,
                minPrice=minPrice,
                maxPrice=maxPrice)
```

Figure 48: image

- If we really need to render Category has a template we could also validade that category is something safe to render.