Hogeschool van Amsterdam

Internet of Things (Indiviual project)

Crypto Alert Security report

Ferran Tombal
500799894

Contents

Introduction

This is a security report for the “Crypto alert” application. Crypto alert is an IoT device that keeps an eye of your savings in the crypto market and alerts users when big profits are made. The embedded device has access the internet and Its being used to obtain data about several cryptocurrencies by calling an external API via the HTTP protocol.
Crypto alert also exists out of a web application. This web application retrieves the cryptocurrency information like the current price from the embedded device using HTTP. It also has a form where users can enter the price and quantity of the cryptocurrency they’ve purchased. Furthermore the website also calculates the total amount of profit/loss and displays all the purchases of the logged in user on the “Purchase” page.
The embedded device retrieves the total amount of profit from the web application and it produces a sound when the user has made large gains and a different sound when users have made a certain amount of loss.

In this study, research will be done that will determine if the Crypto alert application is riddled with any security holes for example which security flaws can be found in the application. The OWASP top 10 list will be used to determine if the product has any vulnerabilities and the specific security threads for IoT devices in general will also appear in this study. Furthermore, some practical research will be done to test the security of the application by using multiple attacks.

The research question of this study is: “How might we research and improve the security and infrastructure of the Crypto alert product so that it meets, for example, the OWASP top 10 analysis requirements.”

Thus the goal of this research is to find vulnerabilities in the application and fix those vulnerabilities so the application will be more secure.

1. Problem analysis

Before solving a specific problem, it is important to correctly understand it. That's where a problem analysis comes into play. Moreover, a problem analysis ensures that you are not solving the wrong problem.

As explained in the introduction, the goal of this study is to find vulnerabilities of the Crypto alert application and fix those vulnerabilities so the application will be more secure.

***Why is this a problem*:**

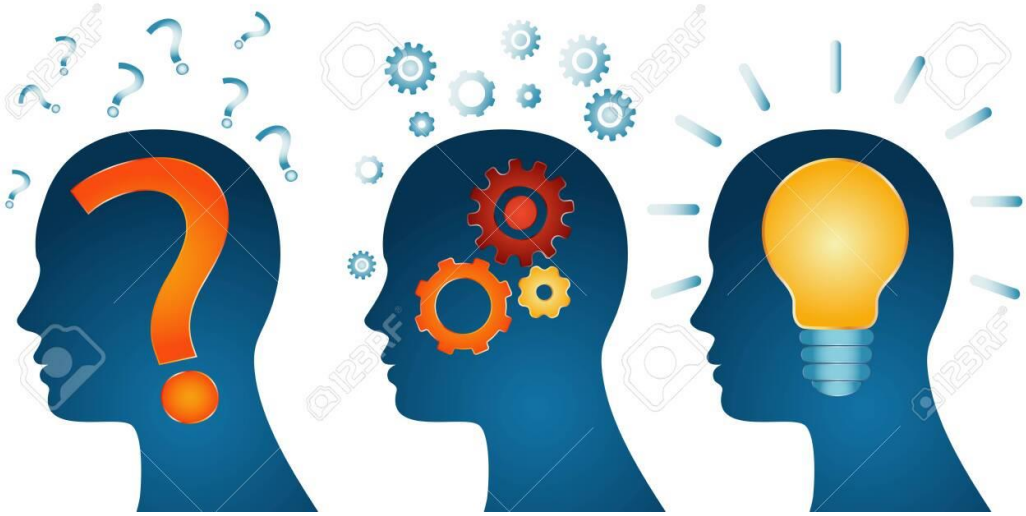
Good security is essential to protect a product from hackers and other malicious users. The OWASP Top Ten is an expert consensus of the most critical risks facing web applications and the teams who are developing them

What happens if it wouldn't' be solved:

Web security is important to keeping hackers and cyber-thieves from accessing sensitive information. Without a proactive security strategy, This product risks the spread and escalation of malware, attacks on other websites, networks, and other IT infrastructures.

To whom is it a problem:

To users using the product or other networks that are connected to the product (Web application or embedded device) in some way.



2. Specific security threads for IoT devices (OWASP top 10)

To find out if the IoT device has any security threads, some vulnerabilities from the OWASP top 10 for IoT devices list (Venafi, n.d) will be analyzed and compared to the Crypto alert IoT application to find any possible vulnerabilities.

1.1 Weak, Guessable, or hardcoded passwords:

"Use of easily bruteforced, publicly available, or unchangeable credentials, including backdoors in firmware or client software that grants unauthorized access to deployed systems."

The IoT device itself doesn't do anything with user passwords or authentication. Users are only able to login on the web application. The web application does not make use of hardcoded passwords since the passwords of the users are stored in a database using the sha256 hash algorithm. On the other hand, the web app allows the use of weak known passwords like for example "password1" which can be brute forced fairly easily (This is called **"Broken authentication"**). A solution could be to implement a functionality so that users can only enter strong passwords when creating an account.

1.2 Insecure network services

"Unneeded or insecure network services running on the device itself, especially those exposed to the internet, that compromise the confidentiality, integrity/authenticity, or availability of information or allow unauthorized remote control. Having insecure network services can lead to DoS, buzzer overflow and fuzzing attacks."

This risk should only be considered when deploying the application to an external server.

1.3 Insecure Ecosystem Interfaces

“Insecure web, backend API, cloud, or mobile interfaces in the ecosystem outside of the device that allows compromise of the device or its related components. Common issues include a lack of authentication/authorization, lacking or weak encryption, and a lack of input and output filtering.”

The embedded devices makes calls to the webserver using HTTP which is unsecure. An idea would be to use the HTTPS protocol to perform requests to the webserver. HTTPS is HTTP with encryption. The difference between the two protocols is that HTTPS uses TLS (SSL) to encrypt normal HTTP requests and responses.

Also, to avoid any unauthorized access to the web API, an “API key” or basic authentication can be used to authenticate the incoming requests. If the client (embedded device in this case) wants to make a HTTP request, it should need to authenticate itself to retrieve certain data.

1.4 Lack of Secure Update Mechanism

“Lack of ability to securely update the device. This includes lack of firmware validation on device, lack of secure delivery (un-encrypted in transit), lack of anti-rollback mechanisms, and lack of notifications of security changes due to updates.”

This is security thread is irrelevant for the Crypto alert application since the project is only for 8 weeks and it won’t get updated after that period.

1.5 Use of insecure and outdated components

“Use of deprecated or insecure software components/libraries that could allow the device to be compromised. This includes insecure customization of operating system platforms, and the use of third-party software or hardware components from a compromised supply chain.”

This does not apply in for the Crypto alert application since I made sure to use most recent versions for most the project dependencies.

1.6 Insufficient Privacy Protection

“User’s personal information stored on the device or in the ecosystem that is used insecurely, improperly, or without permission. Many deployed IoT devices collect personal data that need to be securely stored and processed to maintain compliance with the various privacy regulations, such as GDPR or CCPA. This personal data might be anything from medical information to power consumption and driving behavior. Lack of appropriate controls will jeopardize users’ privacy and will have legal consequences.”

The Crypto alert application collects data like how much total crypto profit or loss a user has made. A possibility could be to check if the data collection meets privacy regulations like the GDRP or CCPA.

1.7 Insecure Data Transfer and Storage

“Lack of encryption or access control of sensitive data anywhere within the ecosystem, including at rest, in transit, or during processing.”

The web application uses sha-256 to hash and store the passwords into the database. The sha-256 algorithm is one of the most secure hashing functions on the market. Agencies of the US government are using sha-256 to protect important sensitive information. It is almost impossible to reconstruct the initial data from the hash value. A brute-force attack would need to make 2^256 attempts to generate the initial data (n-able 2019).

2. Flask specific vulnerabilities

Snyk.io 2021 mentions that there were three known vulnerabilities in older versions of the Flask framework. These were the three direct known vulnerabilities in the flask package of older versions:

Direct Vulnerabilities

Known vulnerabilities in the flask package. This does not include vulnerabilities belonging to this package's dependencies.

Report new vulnerabilities

VULNERABILITY	VULNERABLE VERSIONS	SNYK PATCH	PUBLISHED
H Denial of Service (DoS)	[,0.12.3)	Not available	17 Jul, 2019
H Improper Input Validation	[,0.12.3)	Not available	21 Aug, 2018
L Arbitrary File Download	[,0.6.1)	Not available	14 Sep, 2017

Source: (Snyk.io 2021)

The crypto alert app is using Flask version: "1.1.2" so these vulnerabilities are irrelevant for the application.

The official Flask documentation (Flask, 2021) has a page where all the Flask security considerations are mentioned.

2.1 Cross-Site Scripting (XSS)

The first Flask security consideration is Cross-Site Scripting (XSS). Cross site scripting is the concept of injecting arbitrary HTML into the context of the website. The flask documentation mentions that Flask escapes everything by default so the web application should be secured against XSS attacks.

An important thing to for Flask security are unquoted attributes since there's one thing Flask can't protect you from which is: XSS by attribute injection. So It's necessary to always quote attributes with either double or single quotes when using Jinja expressions in them:

```
<input value="{{ value }}">
```

This is the only HTML attribute in the Crypto alert application that makes use of an Jinja expression:

```
48        <button
49            type="button"
50            class="close"
51            onClick="deletePurchase({{ purchase.id }})"
52        >
```

As you can see, there are quotes around the Jinja expression so Cross-Site Scripting shouldn't be a threat for the application.

2.2 Headers

Browsers recognize various response headers in order to control security. The Flask documentation recommends to use the Flask-Talisman extension to manage HTTPS and the security headers. Flask-Talisman is a small Flask extension that handles setting HTTP headers that can help protect against a few common web application security issues.

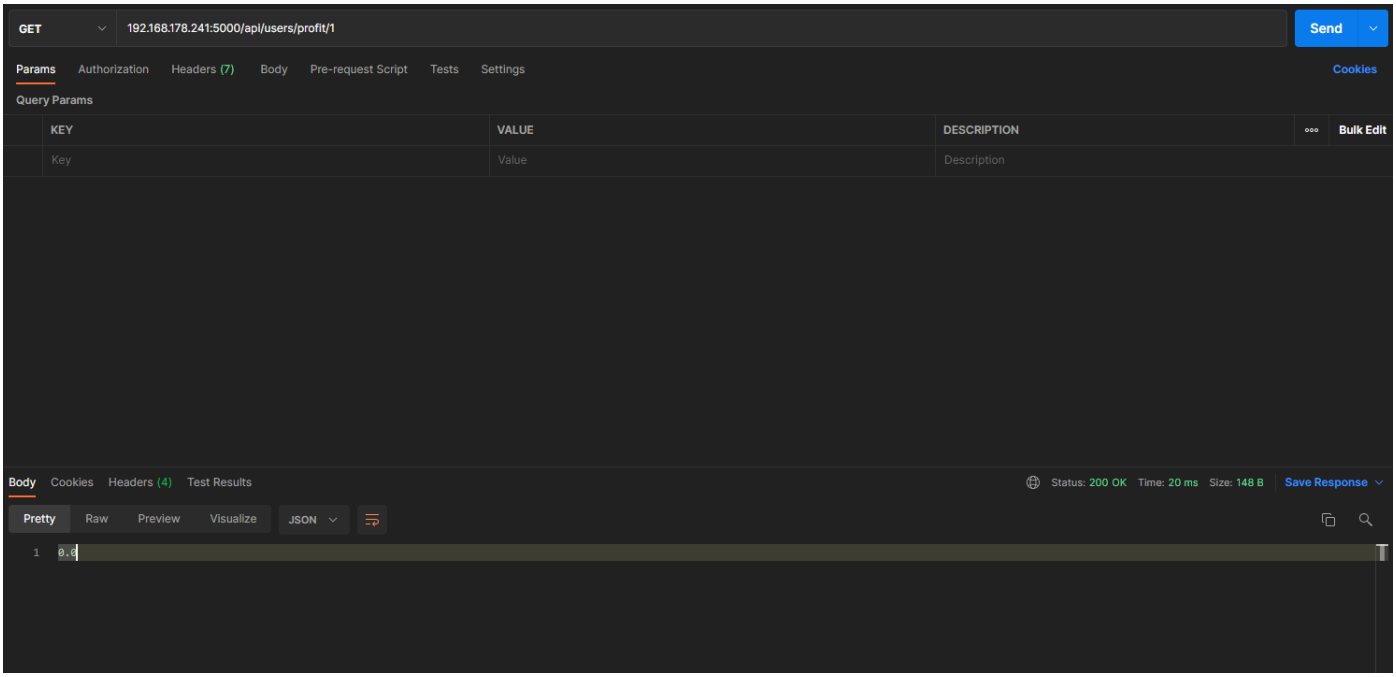
https://github.com/GoogleCloudPlatform/flask-talisman

3. Practical research (Security test of the product)

In this chapter, some practical research (Penetration testing) will be done to further determine the vulnerabilities of the Crypto alert application.

3.1 Sensitive Data Exposure: Unauthenticated API (Man in the middle)

Since the Flask web API doesn't use any authentication, malicious users can easily do a HTTP request to the API to get the total profit of any registered user, to show this, I will do a request HTTP GET request to get a user's profit via Postman without any authentication:



As you can see in the picture above, anyone could just send a HTTP request to the API to get the user's profit of any user by just setting the user ID in the URL.

This corresponds to chapter 1.7 of the OWASP top ten (Insecure data transfer and storage) since the data being transferred from the API is insecure without the usage of any authentication.

3.2 Brute force attack (Testing broken authentication)

A brute force attack is an attack where trial-and-error is being used to guess login info, encryption keys or hidden web pages. Hackers work through all possible combinations hoping to guess correctly. This is an old attack that is still being used to this day (Kaspersy, 2021).

In this subchapter, a brute force attack will be executed on the login page of the web application to find user's login information, this will determine how easy it is to brute force user credentials when a user has an easy guessable password.

An account was created using the following user credentials:

Email: p@gmail.com

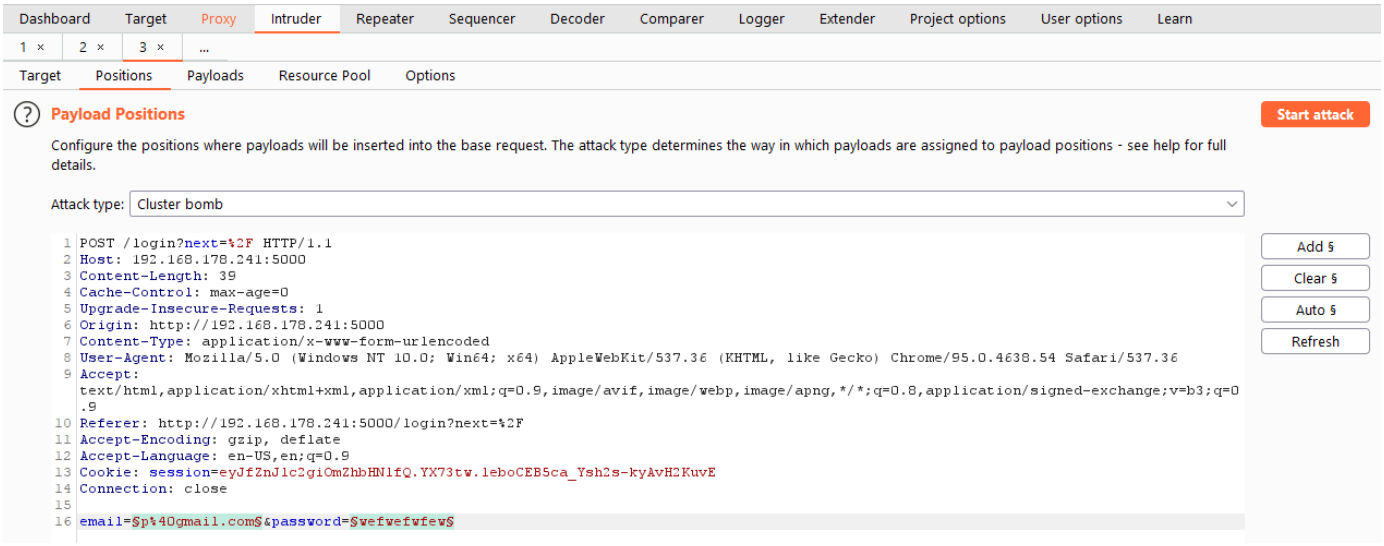
Password: penguin

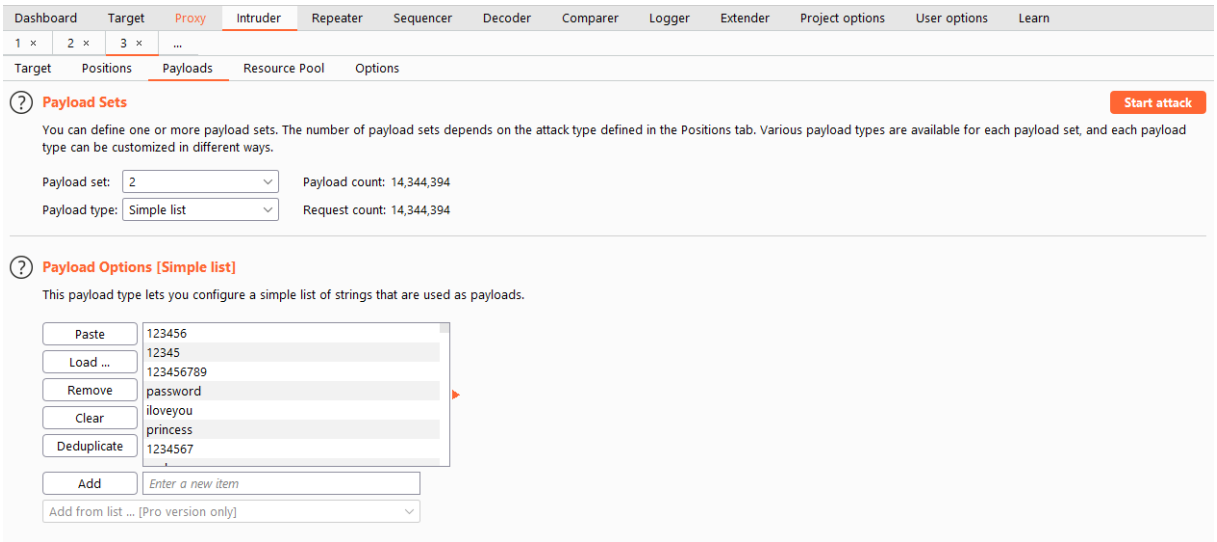
As you can tell, penguin is unsecure password since it doesn't use any special characters, capital letters or numbers.

To brute force a website login you need a word-list, a word-list is a text document containing a list of passwords that are used worldwide.

To brute force the password of p@gmail.com, I used a word-list that was publicly available online which is called "rockyou.txt" and to actually brute force the user password I made use of an application called "Burp Suite".

These are some screenshots of setting up the brute force attack in Burp Suite:

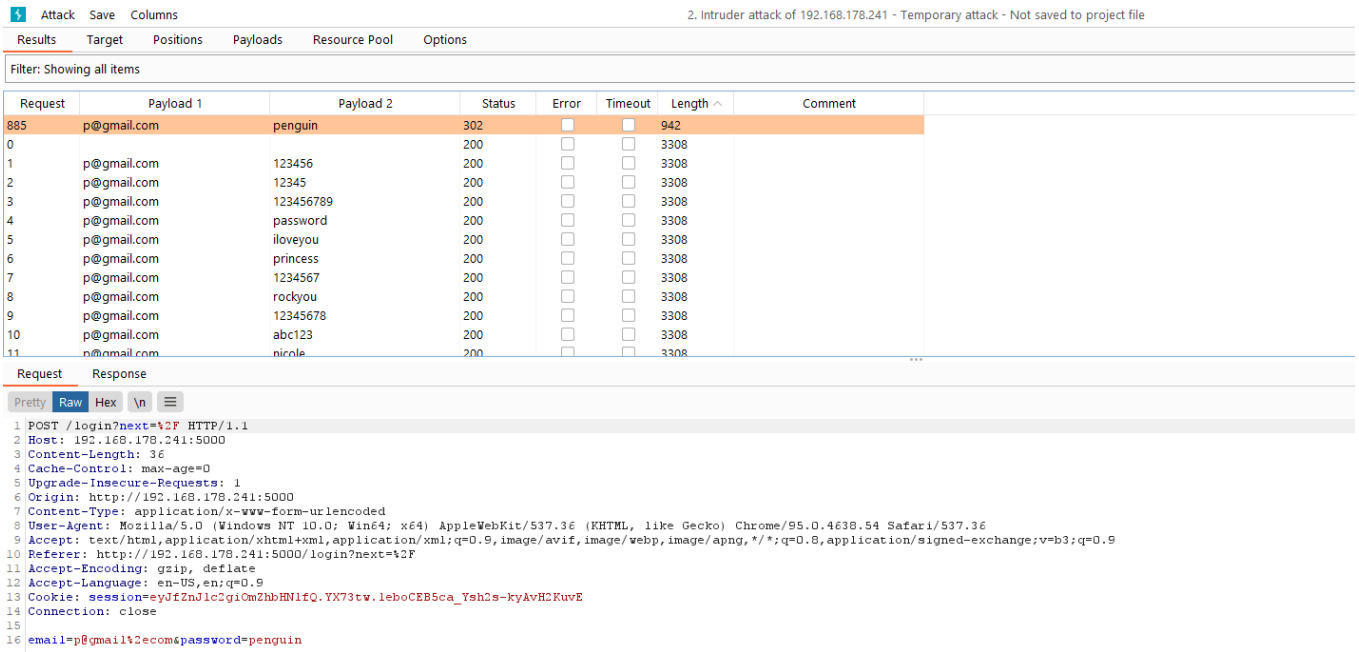




Result:

After approximately an hour of brute forcing the correct password “penguin” was found.

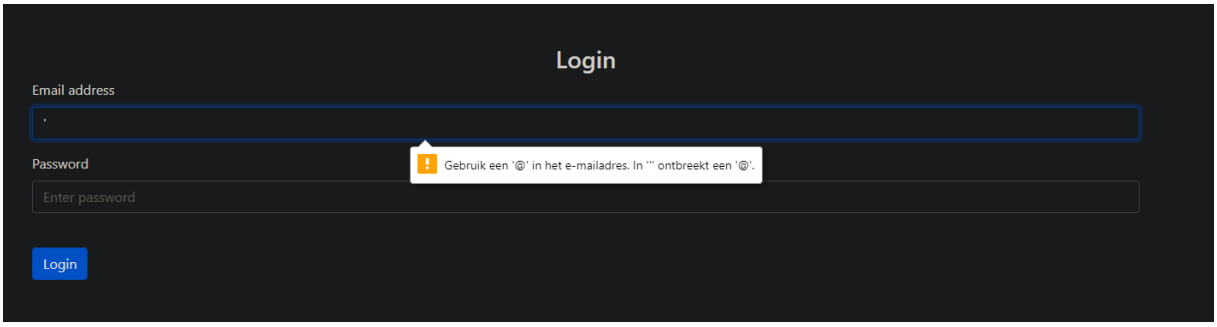
You can tell by the length of the HTTP response (942) that “penguin” is the correct password since all of the other HTTP responses have a length of “3308”:



By looking at the results, weak passwords can easily be brute forced. Therefore a functionality on the registration page of the application needs to be implemented so that only strong passwords are allowed when users are creating an account.

3.3 Testing for SQL Injection (SQLi)

The Crypto alert application shouldn’t be having SQL injection vulnerabilities since the SQL alchemy ORM is being used instead of raw SQL queries. To confirm this a simple ‘ character can be entered in the email input field:



When clicking on the submit button it gives the following error: “Please enter a correct email address”, this is because the input type of the email address is set to ‘email’. Thus is means that the login field isn’t vulnerable to SQL injection attacks.

4. Improving the security of the application

After researching all the vulnerabilities of IoT devices and Flask applications in general, this chapter will mention what exactly is going to be improved.

4.1 Broken authentication (Preventing brute force attacks)

As mentioned in section 1.1, weak passwords can easily be brute forced. To make it more difficult for people to brute force user accounts, I'm going to implement some functionalities so that only strong passwords are allowed when users are creating an account.

Code snippets result:

The password-strength pip library (https://pypi.org/project/password-strength/) was used to enforce strong user password checks in the backend. (*auth.py* file)**:**

```
def sign_up():
    if request.method == 'POST':
        email = request.form.get('email')
        password1 = request.form.get('password1')
        password2 = request.form.get('password2')

        stats = PasswordStats(password1)
        checkpolicy = policy.test(password1)

        user = User.query.filter_by(email=email).first()
        if user:
            flash('Email already exists.', category='error')
        elif len(email) < 4:
            flash('Email must be greater than 3 characters.', ca
tegory='error')
        elif password1 != password2:
            flash('Passwords don\'t match.', category='error')
        elif stats.strength() < 0.30:
            print(stats.strength())
            flash("Password not strong enough. Avoid consecutive
 characters and easily guessed words")
```

Strong password frontend check, in *sign-up.html* file (Using regex):

```
  <div class="form-group">
    <label for="password1">Password</label>
    <input type="password" placeholder="Enter password" class="f
orm-control" name="password1" id="password1" required  pattern="
(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,}" title="Password must contai
n at least one number and one uppercase and lowercase letter, an
d at least 8 or more characters">
```

4.2 Using HTTPS instead of HTTP

Note: This step is only necessary if this product is going to be used as an internal product since most hosting providers already provide HTTPS.

To make sure the data between the client and server is encrypted, HTTPS needs to be used to execute requests instead of HTTP. As explained in section 1.3, The difference between the two protocols is that HTTPS uses TLS (SSL) to encrypt normal HTTP requests and responses. If HTTP requests are made from the client to the server, several attacks can take place like On-path attacks, DNS hijacking, BGP hijacking and Domain Spoofing (Cloudflare, 2021).

Step 1, Create a self-signed certificate using openssl (In powershell):

```
openssl req -x509 -newkey rsa:4096 -nodes -out cert.pem -keyout key.pem -days 365
```

Step 2, Make sure to run the Flask application over HTTPS using the self-signed certificate: To run the Flask app on HTTP, what needed to be done was to use the self-signed certificate in the app.run() call using ssl_context (*main.py file*):

```
from webfiles import create_app

app = create_app()

if __name__ == '__main__':
    app.run('192.168.178.241', ssl_context=('cert.pem', 'key.p
em'), port=5000, debug=True, threaded=False)
```

As you can see in the following picture, the development server now runs on HTTPS instead of HTTP:

```
 * Detected change in 'C:\\Users\\ferra\\documents\\PlatformIO\\Projects\\iot-tom
balf\\Webapplication\\main.py', reloading
 * Restarting with stat
C:\Users\ferra\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.7_qbz5n2k
fra8p0\LocalCache\local-packages\Python37\site-packages\flask_sqlalchemy\__init__
.py:834: FSADeprecationWarning: SQLALCHEMY_TRACK_MODIFICATIONS adds significant o
verhead and will be disabled by default in the future.  Set it to True or False t
o suppress this warning.
  'SQLALCHEMY_TRACK_MODIFICATIONS adds significant overhead and '
 * Debugger is active!
 * Debugger PIN: 151-943-213
 * Running on https://192.168.178.241:5000/ (Press CTRL+C to quit)
```

4.3 Using Basic authorization for the API to avoid unauthorized access

As explained in section 3.1, The “Get user profit” endpoint didn’t make use of any authorization so anyone could get the profit of any other user if they found out the API endpoint.

To fix this vulnerability, I made use of Basic authentication from the flask-HTTPAuth library:

https://flask-httpauth.readthedocs.io/en/latest/

The first thing I had to do in the *cryptoapi.py* file was to create an HTTPBasicAuth object:

```
auth = HTTPBasicAuth()
```

I then had to create a *verify()* method with the *@auth.verify_password* annotation which will check the passed in user email and password from the incoming HTTP request:

```
@auth.verify_password
def verify(email, password):

    count = User.query.filter_by(
        email=email).count()
    if count == 0:
        return False

    user = User.query.filter_by(
    email=email).first()
    if check_password_hash(user.password, password):
        return email
    else:
        return False
```

If user email and password are correct, it will return the email.

On the UserProfit API endpoint I’m then able to protect this endpoint using the *@auth.login_required* annotation so that only when a user email and password are passed it in, it will return the profit of that user. The auth.current_user() call returns the email being returned from the verify() method:

```
class UserProfit(Resource):

    @auth.login_required
    def get(self):
        user = User.query.filter_by(
        email=auth.current_user()).first()
        return user.profit
```

4.4 Security headers

4.4.1 Flask-talisman

I will make use of the Flask-Talisman module which handles HTTP headers that can help protect against a few common web application security issues. Using certain headers can help against MITM (Man In the Middle) attacks, XSS attacks and more.

https://github.com/GoogleCloudPlatform/flask-talisman

First I installed the extension using the pip package manager:

```
pip install flask-talisman
```

I then wrapped the flask app with the Talisman:

```
def create_app():
    app = Flask(__name__)
```

The Flask talisman comes with this default configuration:

The default configuration:

- Forces all connects to `https`, unless running with debug enabled.
- Enables HTTP Strict Transport Security.
- Sets Flask's session cookie to `secure`, so it will never be set if your application is somehow accessed via a non-secure connection.
- Sets Flask's session cookie to `httponly`, preventing JavaScript from being able to access its content. CSRF via Ajax uses a separate cookie and should be unaffected.
- Sets X-Frame-Options to `SAMEORIGIN` to avoid clickjacking.
- Sets X-XSS-Protection to enable a cross site scripting filter for IE and Safari (note Chrome has removed this and Firefox never supported it).
- Sets X-Content-Type-Options to prevent content type sniffing.
- Sets a strict Content Security Policy of `default-src: 'self'`. This is intended to almost completely prevent Cross Site Scripting (XSS) attacks. This is probably the only setting that you should reasonably change. See the Content Security Policy section.
- Sets a strict Referrer-Policy of `strict-origin-when-cross-origin` that governs which referrer information should be included with requests made.

The most important configuration is that is forces all connects to HTTPS which is encrypted instead of HTTP.

4.4.2 Preventing CSRF attacks

To prevent CSRF (Cross-site request forgery) I will use a Flask extension called "Flask-SeaSurf":
It was very simple to use this extension, all I had to do was to install the extension via the pip packet manager and import it into my *__init__.py* file:
I then passed the application object back to the extension like this:

```
csrf = SeaSurf(app)
```

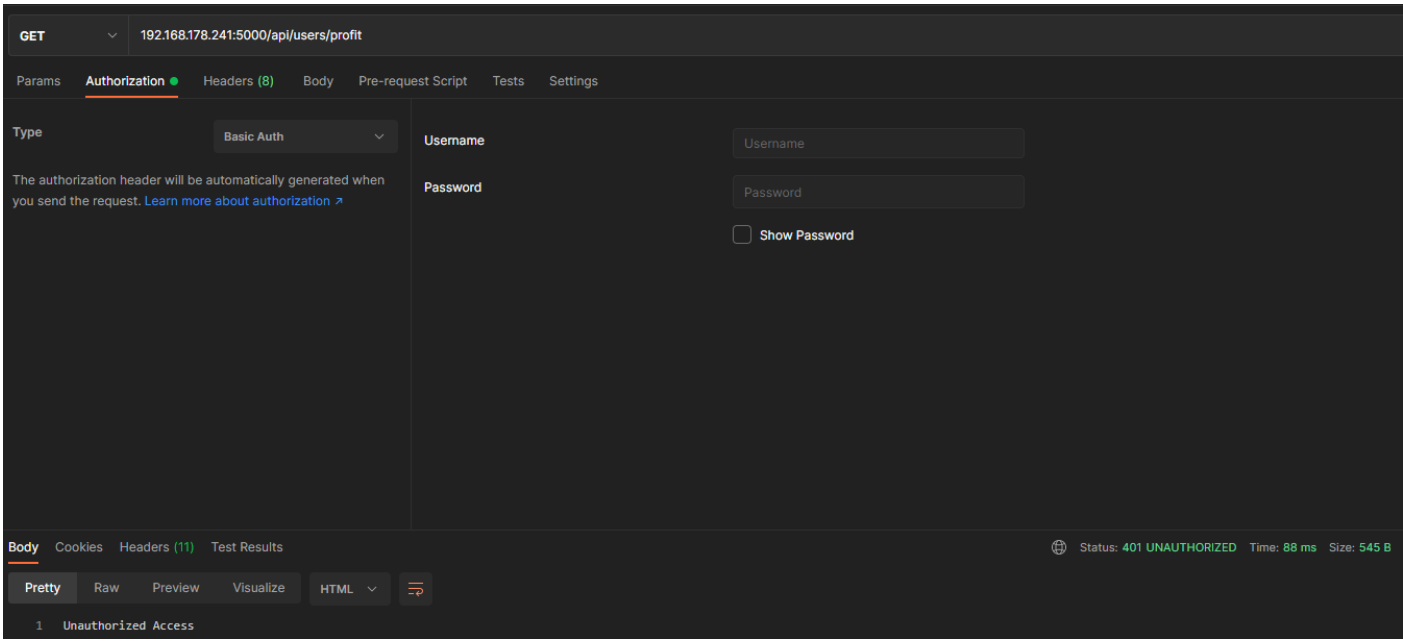
5. Check results (Penetration testing)

In this chapter, some penetration testing will be done to validate the quality of my solutions.

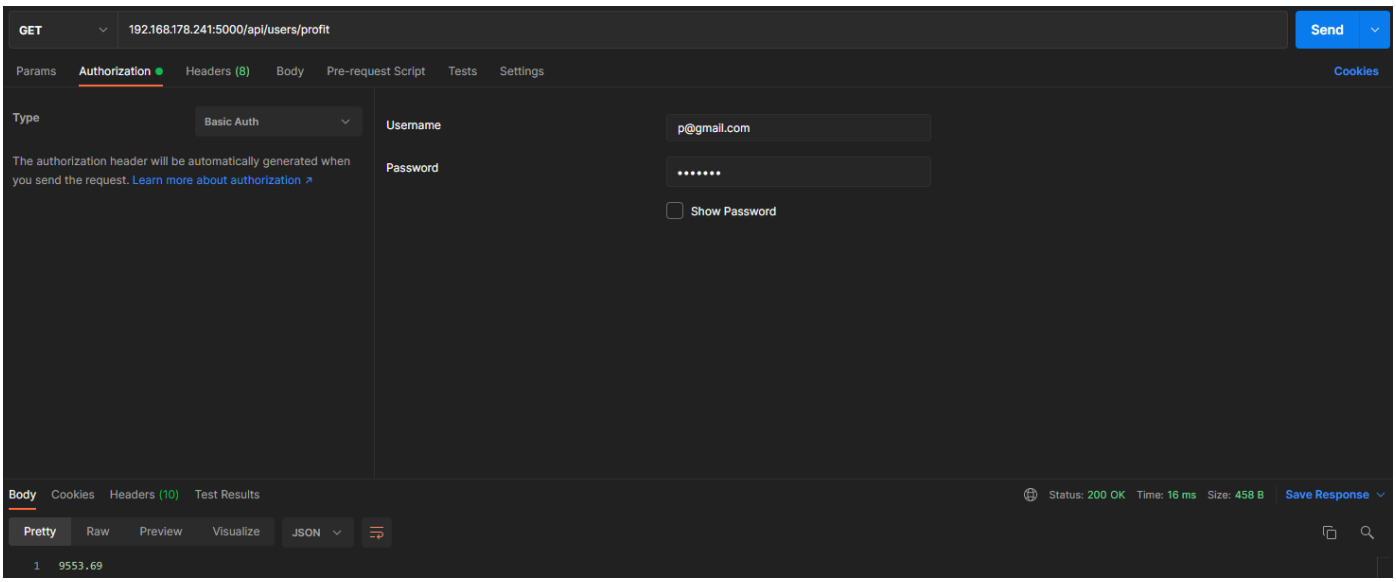
5.1 Testing the authentication of the API

In this subchapter I'm going to do an API call to get a user's profit without passing in any user credentials as the basic Authentication.

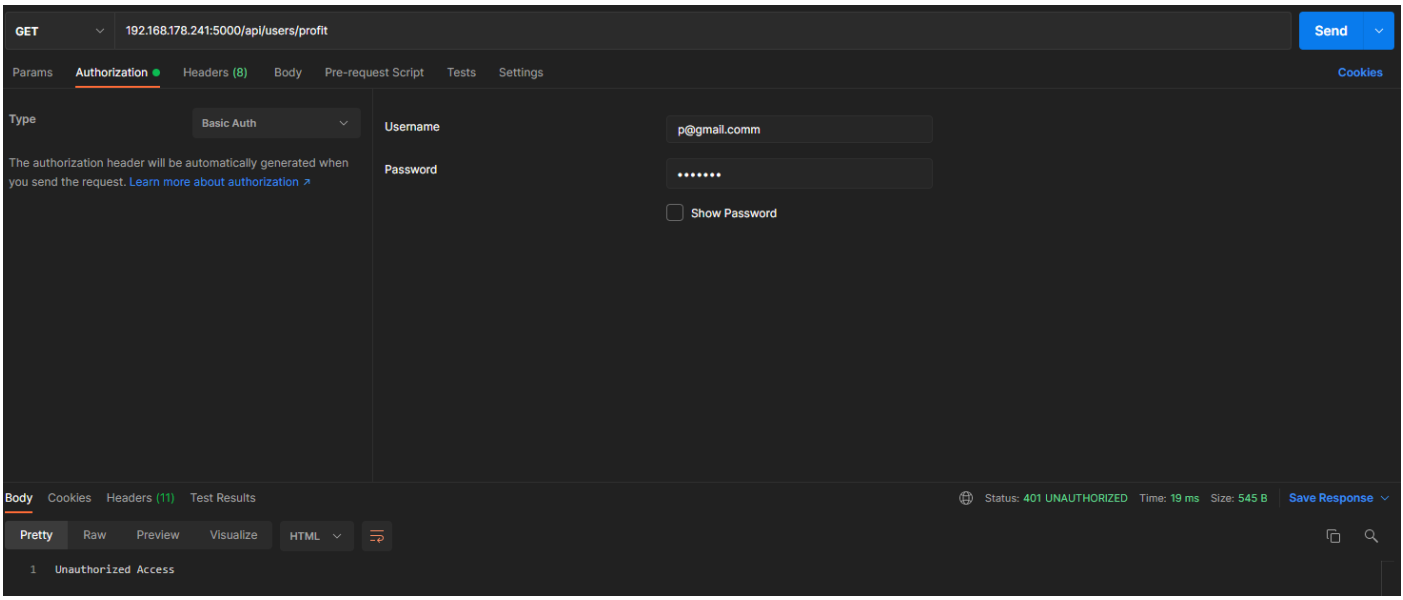
As you can see in Postman, the response status is 401 UNAUTHORIZED:



If I enter the user credentials of a specific user, it does return the profit of that user:



It will also return a 401 UNAUTHORIZED status if someone enters wrong user credentials:

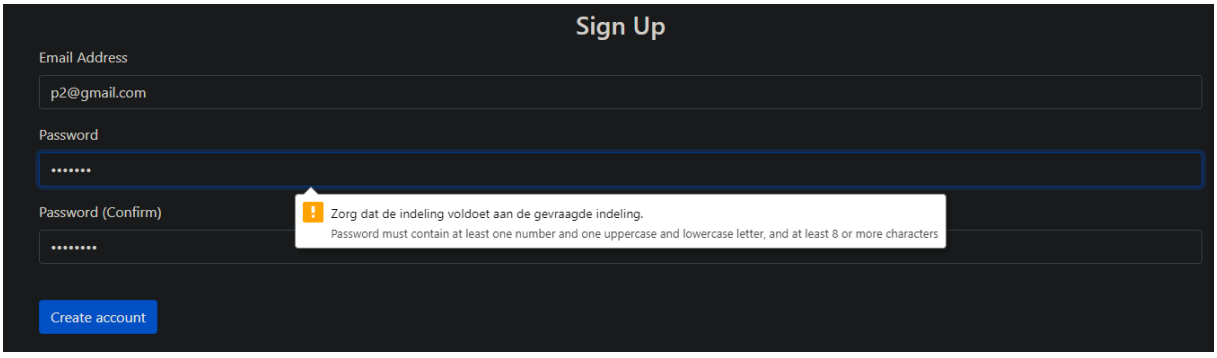


5.2 Brute force

In chapter 3.2 a brute force attack was done to find the password of the user: p@gmail.com which was “penguin”. This could easily be brute forced since that password was in the rockyou.txt word list.

Now two functionalities have been implemented (frontend and backend) so that users can only enter strong passwords.

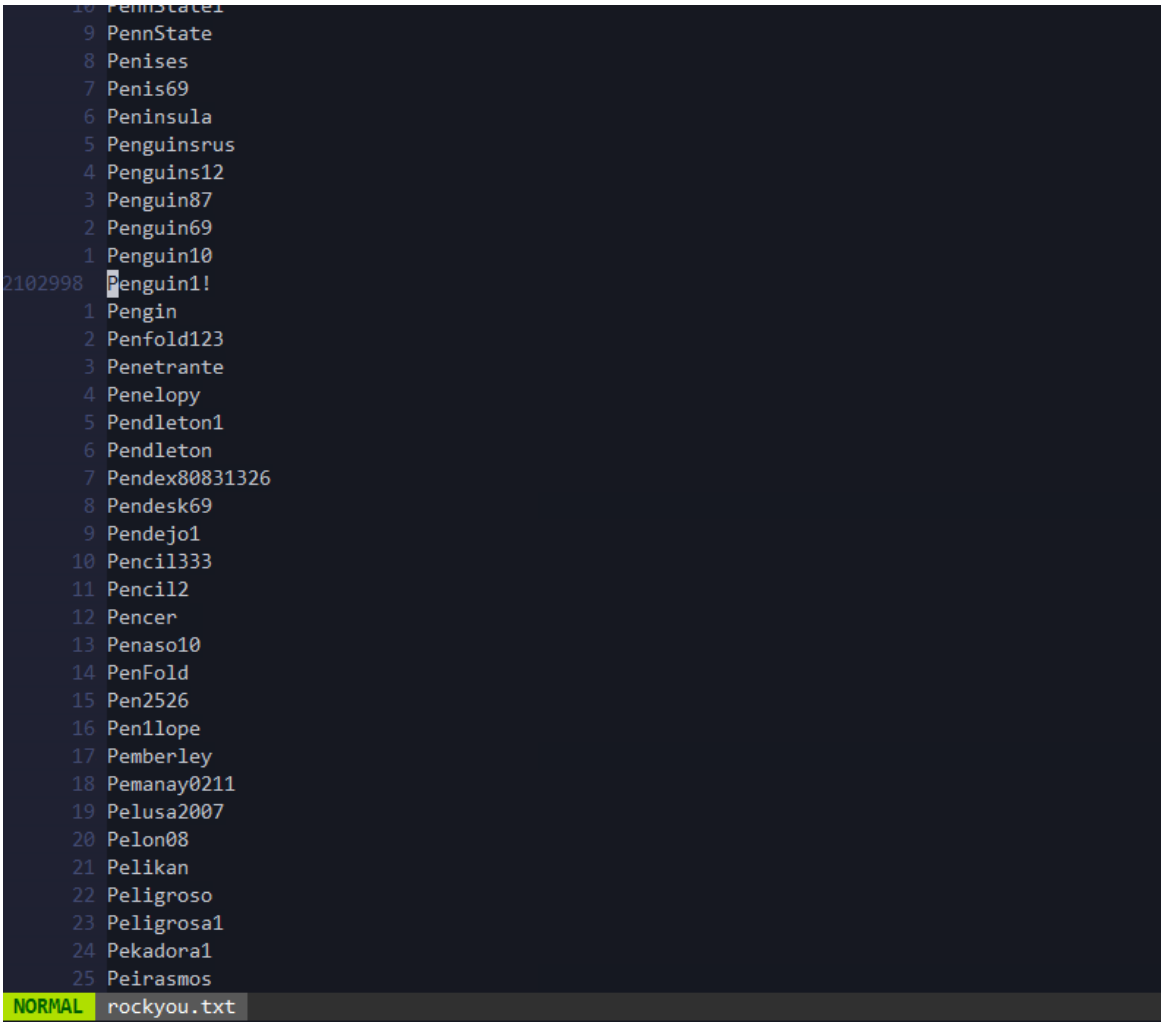
If the user decided to use the password “penguin” it gives the following error message (Password must contain at least one number and one uppercase and lowercase letter, and at least 8 or more characters:



So the user has to follow those strong passwords requirements, the user then for example decides to create a password like this:

Penguin1!

I then searched for “Penguin1!” in the rockyou.txt word list and it was at line 2102998 so that means that it would take a lot longer to brute force “Penguin1!” compared to “penguin”.



Also when the user for example decided to use the password “Penguin5!”, that password doesn’t even show up in the rockyou.txt word list of 14000000000+ passwords:

6. Conclusion

This research sought to answer the question:

“How might we research and improve the security and infrastructure of the product so that it meets, for example, the OWASP top 10 analysis requirements.”

Qualitative research has been carried out into various sources like the OWASP top 10 for IoT devices and Flask security considerations to find possible vulnerabilities of the Crypto Alert application. Also some practical research (Penetration testing) before and after improving the application was done to find vulnerabilities and validate the results.

Looking at the results, the application had some security vulnerabilities. One of them was “Broken authentication”. Users were able to create weak passwords without using any capital letters, numbers or special characters. Therefore the authentication was “broken” and user passwords could be brute forced more easily. Broken authentication has been improved by enforcing strong user password checks in the front and backend.

Furthermore, the “Get user profit” API endpoint didn’t make use of any authentication, therefore anyone could get the profit of any other user if they found out the API endpoint. This vulnerability has been fixed by using Basic Authentication of flask-HTTPAuth library to protect the API endpoint.

Finally, the Flask-Talisman module was used to force all connects to HTTPS instead of HTTP and to manage the security headers.

Sources

Venafi. (n.d). Top 10 vulnerabilities make IoT devices insecure. *Consulted on: https://www.venafi.com/blog/top-10-vulnerabilities-make-iot-devices-insecure*

n-able. (2019, 12th September). SHA-256 Algorithm Overview. *Consulted on: https://www.n-able.com/blog/sha-256-encryption*

Flask. (2021). Security considerations. *Consulted on: https://flask.palletsprojects.com/en/2.0.x/security/*

Cloudflare. (2021). Why is HTTP not secure. *Consulted on: https://www.cloudflare.com/learning/ssl/why-is-http-not-secure/*

Kapersky. (2021). Brute force attack. *Consulted on: https://www.kaspersky.com/resource-center/definitions/brute-force-attack*