

CS 437 Assignment 1

Anıl Ozan Ayhan – 26565

Mert Ali Çelik – 26958

<https://drive.google.com/drive/folders/10dyYYZ6txgKAsKbmQgY-lEy0euxHvO3I?usp=sharing>

Introduction

This PDF explains our CS437 Assignment 1, creating a Turkish travel website honeypot. We explain each step of our task below, with a detailed look at our methods, tools and workflow.

Implementation Details

Code Structure and Logic

The `app.py` script serves as the central component of the Flask application. It handles web requests, user authentication, database interactions, and security features. We have created all the endpoints necessary to create a real travel news website. Our RSS feed is from NTV's travel news.

Database Creation and Population

The SQLite database is used for storing user credentials, comments, and administrative data. It is interacted with through `sqlite3`, which allows for database management within Python. We insert several accounts into admin and users tables to test our app with.

```
users = [  
    ('mertali@example.com', '12345678'),  
    ('anilayhan@sabanciuniv.edu', '12345678'),  
    ('asd@mail.com', '12345678'),  
    ('mertali@sabanciuniv.edu', '12345678')  
]
```

```
admins = [  
    ('mertali@example.com', '12345678'),  
    ('anilayhan@sabanciuniv.edu', '12345678')  
]
```

Explanation of Imports

Flask-related Imports

- `from flask import Flask, request, render_template, session, redirect, url_for, send_file, render_template_string, flash`
 - These imports from Flask are essential for web application functionalities such as request handling, URL routing, HTML rendering, and session management.

Security and Utility Imports

- `import sqlite3`
 - Enables interaction with a SQLite database, which is used for storing application data.
- `import hashlib`
 - Provides hashing functions, crucial for securely storing and verifying passwords.
- `import random`
 - Used to generate random numbers, for instance, in creating CAPTCHA challenges and reset codes.
- `import datetime`
 - Necessary for handling dates and times, especially for setting expiration times for tokens and codes.

Email and CAPTCHA Handling Imports

- `from flask_mail import Mail, Message`
 - Flask extension for sending emails, used in password reset and multi-factor authentication processes.
- `from captcha.image import ImageCaptcha`
 - Generates image-based CAPTCHAs to protect against automated form submissions.

Logging and Feed Parsing Imports

- `import logging`
- `from logging.handlers import RotatingFileHandler`
 - These imports are for logging requests and actions within the application, useful for monitoring activities and potential security breaches.
- `import feedparser`
 - Parses RSS feeds, which allows the application to display dynamic content fetched from external news sources.

SMS Communication Imports

- `from twilio.rest import Client`
 - This import is for the Twilio client, which enables the application to send SMS messages as part of the multi-factor authentication for admin users.

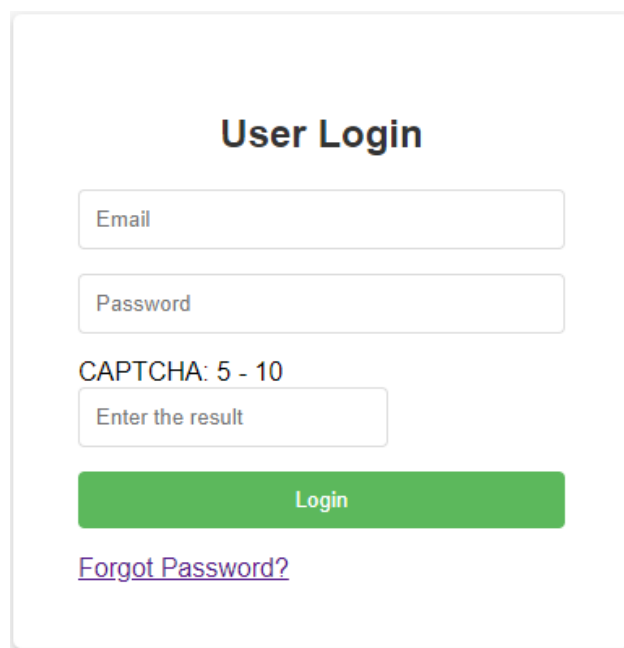
Code Analysis and Security Check Imports:

- `pip install prospector`
 - A tool for static code analysis that examines various aspects of your codebase, providing insights into coding style, errors, and more.
- `pip install bandit`
 - A security linter that checks for common security issues in Python code, helping to identify potential vulnerabilities.
- `pip install mypy`
 - A static type checker for Python that aims to identify and eliminate common programming errors related to typing.

Vulnerabilities and Protections Demonstration

Our application is designed to demonstrate specific vulnerabilities and their respective protection mechanisms. Each vulnerability, such as the lack of rate limiting or a weak CAPTCHA, is explained with an example of how an attacker could exploit it. Corresponding protections, like multi-factor authentication via SMS, are also demonstrated.

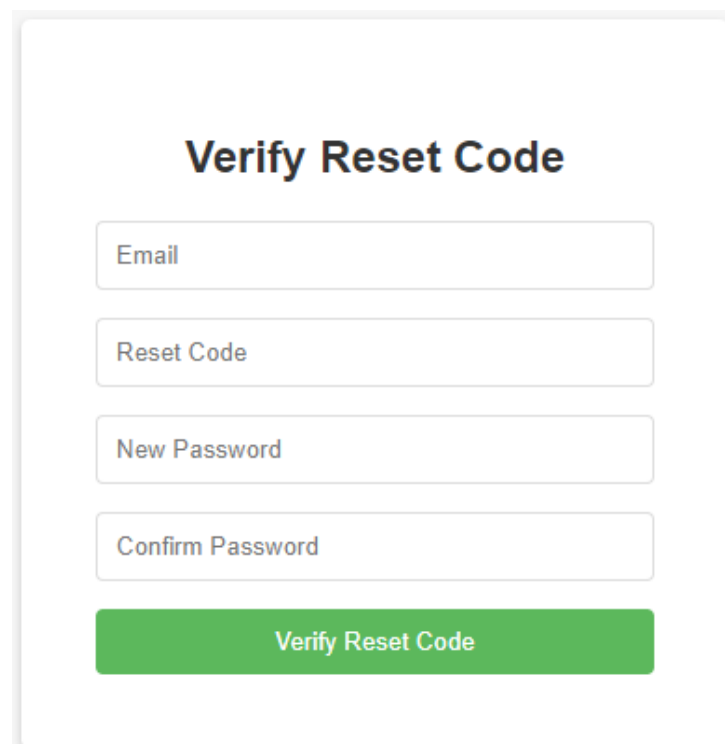
Our first vulnerability is weak captcha for user login.



The image shows a 'User Login' form with a white background and a light gray border. At the top, the title 'User Login' is centered in a bold, black font. Below the title, there are three input fields: 'Email', 'Password', and 'CAPTCHA: 5 - 10'. The 'Email' and 'Password' fields are white with a light gray border. The 'CAPTCHA' field is a white box with the text 'Enter the result' inside. Below the input fields is a green 'Login' button with white text. At the bottom, there is a purple link that says 'Forgot Password?'.

As we can see in the picture, the weak captcha is a simple operation. The left and right sides of our operands are numbers from 1 to 10. And there will be one of the four basic operations in the middle. Even though this is randomized, it is still very easy for an attacker to get through this captcha every time. The picture below shows our breakcaptcha script finding and getting through the captcha.

```
PS C:\Users\anilo\OneDrive\Masaüstü\cs437hw> python breakcap.py
Found captcha: 7 + 5
Answered: 12
Login successful!
```



Our second vulnerability is the forgot password functionality. This form requires users mail address, 2 digit reset code sent to their mail and their new password. The weakness of this code is that there is no rate limiting for the reset code. It is also just 2 digits. Even a person not using any scripts has a 1/100 chance of guessing the correct code and resetting the password. Below are our images showing the received reset code mail and our brute force code finding the correct code.



mertalick13@gmail.com

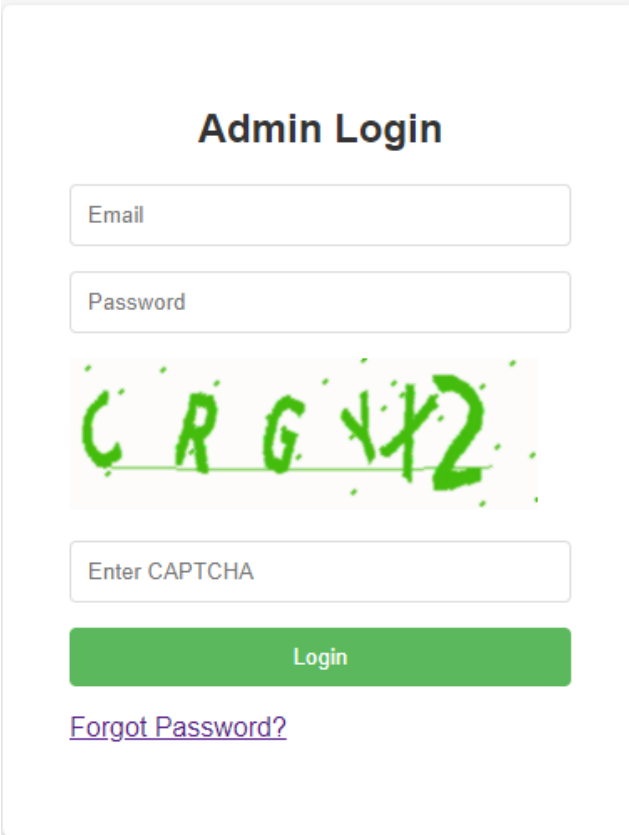
21:15 (2 minutes ago)



to me ▼

Your password reset code is: 57 This code will expire in 2 minutes.

Our admin side however, has no problems with these issues. We have a strong randomized captcha consisting of letters and numbers. The text is also distorted, fonts randomized, characters can overlap and colors change every time. This makes it hard for an attacker to get through.

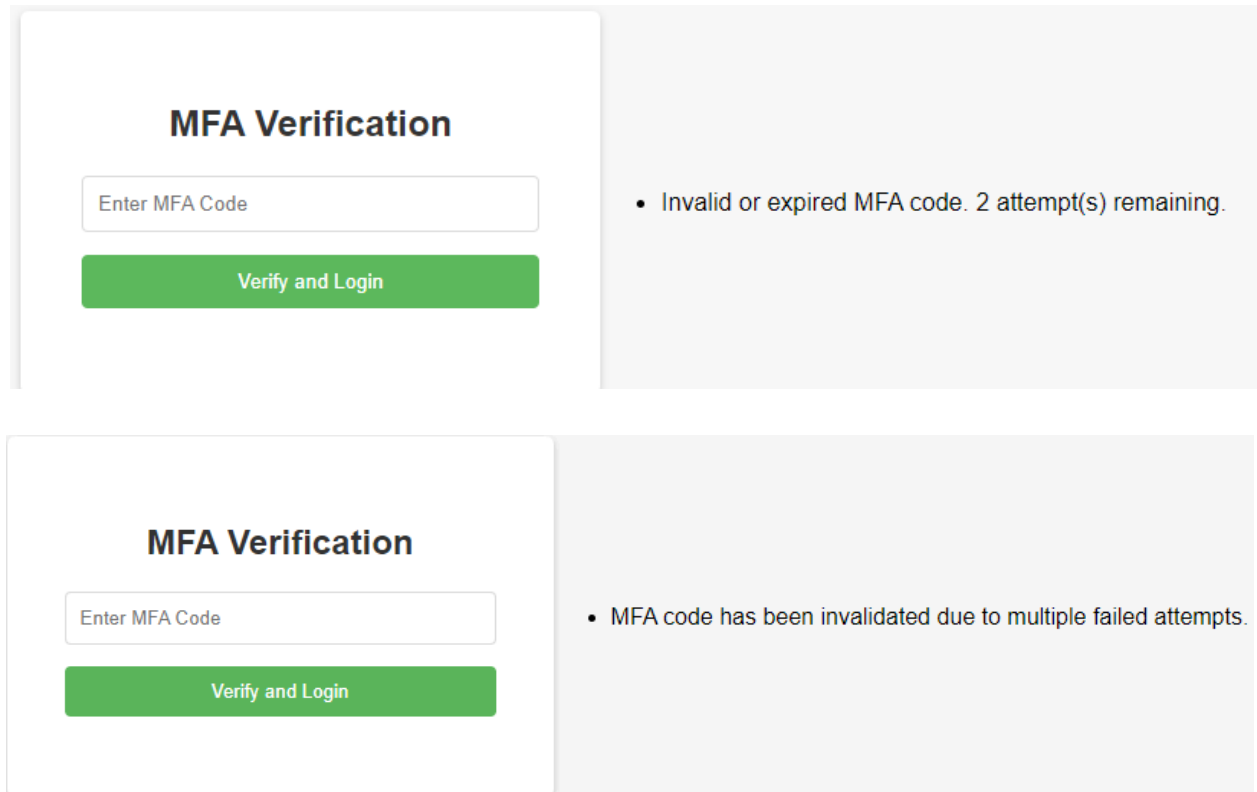


The image shows a web form titled "Admin Login". It contains three input fields: "Email", "Password", and "Enter CAPTCHA". Below the "Enter CAPTCHA" field is a green "Login" button. At the bottom of the form is a link labeled "Forgot Password?". The CAPTCHA image displays the text "C R G 1 4 2" in a green, distorted, and pixelated font.

Our captcha breaker below can be seen failing to find a captcha and make a login request.

```
PS C:\Users\anilo\OneDrive\Masaüstü\cs437hw> python breakadmin.py
CAPTCHA not found.
Failed to make login request.
```

We also can not brute force the admin verification code. The verification code for admins is 6 digits and the user can only try 3 times before the code is invalidated. This makes it highly unlikely that an attacker will get the correct code and log in with admin privileges.



The pictures show the 3 tries the user has before multi factor authentication code is invalidated.

Static Code Analysis

Tools like Bandit, PYT, Rough-Auditing-Tool-for-Security, and Prospector were used to analyze the code for common security issues. The effectiveness of these tools is discussed with detailed findings.

We used 4 tools for static analysis. The first was **Prospector**. It showed us some important details. We had:

- Unused imports
- Unnecessary else statements
- Use of eval (security risk)
- Unused variables

We had a total of 19 messages received from prospector.

Second tool we used was **Bandit**. It showed some of the same issues as prospector but also notified us of these (different) issues:

- Hardcoded passwords
- Insecure use of random

The third static analyzer we used was **Mypy**. It showed us:

- Need type annotation

Mypy gave a small amount of data compared to the other analyzers. It just showed us that we needed to provide explicit type annotations for some variables such as `admin_mfa_codes`, `comments` and `admin_attempt_counters` which were empty lists.

The last tool we used was **Pylint**. It gave our code a 7.4 out of 10. The unique issues it gave us were:

- Lines too long
- Trailing whitespace
- Broad exception caught
- Unspecified encoding
- Reimported and wrong import position

We used these tools in succession one after another without fixing the issues they presented. Our reasoning for this was so that we could see how different they evaluate the same code.

We also used **SQLMap** to check our code for injection possibilities. It first gave us a few warnings. These warnings said that the CAPTCHA we had and page refresh with each request could make it hard for SQLMap. The logs showed that a wide range of injection techniques were used but there were no successful injection vulnerabilities. Since we used a randomised captcha, it was not possible to get through it and test SQLmap fully.

Monitoring and Logging

The honeypot's ability to monitor each vulnerability abuse and access point is showcased in our `/monitor` page. It shows different values depending on different requests. An example screenshot is placed below.

Time: 2024-01-10 21:16:56, RequestType: POST, Endpoint: /verify_reset_code, Status: Password Reset Abuse, IP: 127.0.0.1, UserAgent: python-requests/2.31.0, AttemptType: Normal Request

Time: 2024-01-10 21:16:56, RequestType: POST, Endpoint: /verify_reset_code, Status: Password Reset Abuse, IP: 127.0.0.1, UserAgent: python-requests/2.31.0, AttemptType: Normal Request

Time: 2024-01-10 21:16:56, RequestType: POST, Endpoint: /verify_reset_code, Status: Password Reset Abuse, IP: 127.0.0.1, UserAgent: python-requests/2.31.0, AttemptType: Normal Request

Time: 2024-01-10 21:16:56, RequestType: POST, Endpoint: /verify_reset_code, Status: Normal Request, IP: 127.0.0.1, UserAgent: python-requests/2.31.0, AttemptType: Normal Request

Admin Login Attempts

RequestType: POST, Endpoint: /admin_login, Status: Failed, IP: 127.0.0.1, UserAgent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0, AttemptType: Admin Login Attempt

RequestType: POST, Endpoint: /admin_login, Status: Failed, IP: 127.0.0.1, UserAgent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0, AttemptType: Admin Login Attempt

RequestType: POST, Endpoint: /admin_login, Status: Failed, IP: 127.0.0.1, UserAgent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0, AttemptType: Admin Login Attempt

RequestType: POST, Endpoint: /admin_login, Status: Failed - Incorrect CAPTCHA, IP: 127.0.0.1, UserAgent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0, AttemptType: Admin Login Attempt

Time: 2024-01-09 00:33:19, RequestType: POST, Endpoint: /login, Status: Attempt, IP: 10.65.5.249, UserAgent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36, AttemptType: Normal Request

Time: 2024-01-09 00:36:33, RequestType: POST, Endpoint: /login, Status: Attempt, IP: 127.0.0.1, UserAgent: python-requests/2.31.0, AttemptType: Normal Request

Time: 2024-01-09 00:36:52, RequestType: POST, Endpoint: /login, Status: Attempt, IP: 127.0.0.1, UserAgent: python-requests/2.31.0, AttemptType: Normal Request

Time: 2024-01-10 21:13:33, RequestType: POST, Endpoint: /login, Status: Attempt, IP: 127.0.0.1, UserAgent: python-requests/2.31.0, AttemptType: Normal Request

Verify Reset Code Attempts

RequestType: POST, Endpoint: /verify_reset_code, Status: Attempt, IP: 127.0.0.1, UserAgent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0, AttemptType: Normal Request

RequestType: POST, Endpoint: /verify_reset_code, Status: Attempt, IP: 127.0.0.1, UserAgent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0, AttemptType: Normal Request

RequestType: POST, Endpoint: /verify_reset_code, Status: Attempt, IP: 127.0.0.1, UserAgent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0, AttemptType: Normal Request

Team Responsibilities and Contributions

Mert Ali and Anıl worked with Extreme Programming (XP) frame methodology to complete this project. Mert Ali and Anıl completed tasks by talking virtually over many hours according to the timeline for this project. We informed each other after completing tasks because we need to work in pairs to complete this project. Also we divided User Page and Admin User Page as two separate jobs. Mert Ali designed the User Page authentication page. Anıl designed the Admin Page.

Team Member	Mert Ali Çelik	Anıl Ozan Ayhan
Responsibilities	User Page authentication	Admin Page authentication
Responsibilities	Database creation	Database creation
Responsibilities	Home Page creation	RSS feature addition
Responsibilities	email-verification for forgot password user	sms-verification for forgot password admin
Responsibilities	vulnerability tests	brute force code for testing
Responsibilities	html implementation	Captcha breaking script
Responsibilities		Logout

Ömer didn't contribute to the project.

Conclusion

It was a fun and challenging project. We successfully simulated a web environment with both vulnerabilities and security measures. The honeypot serves as a valuable educational tool for understanding cybersecurity in web applications.