**DETECTING AND PREVENTING HTTP DESYNC ATTACKS IN WEB SERVER**

**A PROJECT REPORT**

***Submitted by***

**SUJITH KUMAR K (190071601162)**

**VENKATESAN P (190071601172)**

**Under the guidance of**

**DR. S. REVATHI**

**Professor**

***In partial fulfillment for the award of the degree of***

**BACHELOR OF TECHNOLOGY**

*in*

**COMPUTER SCIENCE AND ENGINEERING**



**MAY 2023**



**BONAFIDE CERTIFICATE**

Certified that this project report “**DETECTING AND PREVENTING HTTP DESYNC ATTACKS IN WEB SERVER**” is the bonafide work of “**SUJITH KUMAR K (190071601162) AND VENKATESAN P (190071601172)** who carried out the project work under my supervision. Certified further, that to the best of our knowledge the work reported herein does not form part of any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

*SIGNATURE* *SIGNATURE*

**Dr. S. Revathi** **Dr. Aisha Banu W**

Professor Head of the Department

Department of CSE Professor & Head

B.S. Abdur Rahman Crescent B.S. Abdur Rahman Crescent

Institute of Science and Technology Institute of Science and Technology Vandalur, Chennai – 600 048 Vandalur, Chennai – 600 048



**VIVA VOCE EXAMINATION**

The viva voce examination of the project work titled **“DETECTING AND PREVENTING HTTP DESYNC ATTACKS IN WEB SERVER”** submitted by **SUJITH KUMAR K (190071601162) and VENKATESAN P (190071601172)** is held on \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

## INTERNAL EXAMINER EXTERNAL EXAMINER

**ACKNOWLEDGEMENT**

We sincerely express our heartfelt gratitude to **Prof. Dr. T. MURUGESAN,** Pro-Vice Chancellor, B.S. Abdur Rahman Crescent Institute of Science and Technology, for providing us an environment to carry out our course successfully.

We sincerely thank **Dr. N. RAJA HUSSIAN**, Registrar for furnishing every essential facility for doing our project.

We thank **Dr. SHARMILA SANKAR**, Dean, School of Computer, Information and Mathematical Sciences for her motivation and support.

We thank **Dr. AISHA BANU W,** Professor and Head, Department of Computer Science and Engineering, for providing strong oversight of vision, strategic direction and valuable suggestions.

We express our sincere thanks to the Project Review Committee members, from the Department of Computer Science and Engineering, **Dr. S. REVATHI**, Professor and

**Dr. MADHINA BANU,** Assistant Professor (Sr. Gr.) for their valuable suggestions and support.

We obliged our project supervisor **Dr. S. REVATHI,** Professor, Department of Computer Science and Engineering for her professional guidance and continued assistance during our project.

We thank our class advisor, **Dr. S.P. VALLI,** Assistant Professor (Sel. Gr), Department of Computer Science and Engineering for her guidance and encouragement throughout the internship period.

We thank all the **Faculty members** and the **System staff** of the Department of Computer Science and Engineering for their valuable support and assistance at various stages of project development.

## SUJITH KUMAR K

## VENKATESAN P

**ABSTRACT**

The HTTP Desync Attack represents a significant security vulnerability that jeopardizes the integrity of web applications and servers that rely on the HTTP protocol. Exploiting a discrepancy in the interpretation of HTTP protocol messages between the client and server, attackers gain the ability to manipulate these messages, potentially compromising the security of the targeted web application. HTTP Desync attack involves downgrading from the more secure and advanced HTTP/2 protocol to the older and less secure HTTP/1.1 protocol. Security researcher James Kettle made notable contributions to the field by investigating various techniques that can be deployed to exploit the vulnerability.

Kettle's research findings demonstrated how malicious actors can send carefully crafted HTTP requests, inducing the client and server to revert to the outdated protocol. The downgrade facilitates the execution of diverse malicious activities, including code injection and unauthorized access to sensitive information. To effectively mitigate the specified vulnerability, web developers and server administrators must ensure the appropriate configuration of their web applications and servers, enabling support for the secure HTTP/2 protocol and preventing any downgrade to older, less secure protocols. Additionally, implementing a range of security measures, such as comprehensive input validation and robust output encoding, can significantly bolster defenses against potential HTTP Desync Attacks.

# TABLE OF CONTENTS

|  |  |  |  |
| --- | --- | --- | --- |
| **CHAPTER NO.** | | **TITLE** | **PAGE**  **NO.** |
|  | | **ABSTRACT** | **iv** |
|  | | TABLE OF CONTENTS | **v** |
|  | | **LIST OF FIGURES** | **viii** |
|  | | **LIST OF TABLES**  **LIST OF ABBREVIATIONS** | **ix**  **x** |
| **1** | | **INTRODUCTION** | 1 |
| 1.1 | | GENERAL | 1 |
| 1.1.1 | | OVERVIEW | 1 |
| 1.1.2 | | DESCRIPTION | 2 |
| 1.1.3  1.2  1.3  1.3.1  1.3.2  1.3.3  1.4  1.5  1.6  1.6.1  1.6.2  1.7  1.8  1.8.1  1.8.2  1.9  1.10  1.10.1  1.11 | | OBJECTIVE  ABOUT THE PROJECT  HTTP – BASICS  HTTP/1.0  HTTP/1.1  HTTP/2  HTTP REQUEST  HTTP RESPONSE  HTTP METHODS  HTTP HEADERS  COMMON HTTP STATUS CODES AND DESCRIPTION  HOW A SERVER HANDLES REQUEST AND RESPONSE  CACHING  SERVER-SIDE CACHING  CLIENT-SIDE CACHING  UNDERSTANDING THE COMPONENTS OF A  WEB SERVER: AN OVERVIEW OF FRONT-END  AND BACK-END DEVELOPMENT  HTTP DESYNC ATTACK AND TYPES  H2.CL  FLOW DIAGRAM | 2  3  3  4  4  5  6  7  8  9  11  12  14  14  15  16  17  20  21 |
| **2** | | **LITERATURE SURVEY** | **24** |
| **3** | | **PROBLEM DEFINITION & METHODOLOGIES** | **28** |
| 3.1 | | PROBLEM DEFINITION | 28 |
| 3.2 | | EXISTING SYSTEM | 28 |
| 3.2.1 | | EXISTING SYSTEM TECHNIQUE | 28 |
| 3.2.2 | | ISSUES IN EXISTING SYSTEM | 29 |
| 3.3  3.3.1  3.4  3.4.1  3.4.2  3.4.3 | | PROPOSED SYSTEM  ADVANTAGES OF EXISTING SYSTEM  COMPARISON AND ANALYSIS  EFFICIENCY  BREAKDOWN OF HTTP PROTOCOL BASED  ON USAGE  COMPARATIVE ANALYSIS OF TOP WAFs USED | 29  30  30  30  32  35 |
| **4** | | **DESIGN PROCESS** | **36** |
| 4.1 | | SYSTEM REQUIREMENTS | 36 |
| 4.1.1 | | HARDWARE REQUIREMENTS | 36 |
| 4.1.2 | | SOFTWARE REQUIREMENTS | 36 |
| 4.1.2.1 | | DOCKER | 36 |
| 4.1.2.2 | | BURPSUITE | 37 |
| 4.1.2.3 | | WEB BROWSER | 38 |
| 4.1.2.4 | | TURBO INTRUDER | 39 |
| 4.2 | | MODULE EXPLANATION | 40 |
| 4.2.1  4.2.2  4.2.3  4.2.4  4.2.5  4.3 | | DEPLOYMENT  ANALYSIS  DEMONSTRATION  PROTECTION  UPDATES  SYSTEM ARCHITECTURE | 41  41  41  42  42  43 |
| **5**  **6**  **6.1**  **6.2**  **7**  **8**  **9**  **10** | | **IMPLEMENTATION**  **CONCLUSION AND FUTURE ENHANCEMENT**  **CONCLUSION**  **FUTURE ENHANCEMENT**  **REFERENCES**  **APPENDIX**  **A1 – SOURCE CODE**  **A2 – SCREENSHOTS**  **TECHNICAL BIOGRAPHY**  **PLAGIARISM CHECKING** | **46**  **47**  **47**  **47**  **48**  **50**  **50**  **69**  **76**  **78** |
|  |
|  |
|  |
|  |

**LIST OF FIGURES**

|  |  |  |
| --- | --- | --- |
| **FIGURE NO.** | **TITLE** | **PAGE.NO** |
| Fig. 1.1 | Standard HTTP Request | 5 |
| Fig. 1.2 | Standard HTTP Response | 6 |
| Fig. 1.3 | Twitter HTTP/1.1 request | 7 |
| Fig. 1.4 | Twitter HTTP/2 request | 8 |
| Fig. 1.5 | Server - Side Caching | 15 |
| Fig. 1.6 | Client - Side Caching | 15 |
| Fig. 1.7 | Working of a Server | 17 |
| Fig. 1.8 | HTTP Desync attack mechanism | 18 |
| Fig. 1.9 | Proposed System Workflow | 21 |
| Fig. 3.1  Fig 3.2  Fig 3.3  Fig 4.1 | Efficiency Comparison of HTTP Protocols: HTTP 1.0, 1.1, and 2.0  Comparison of HTTP Protocols based on usage  Top WAFs used in the Industry  System Architecture | 31  32  34  38 |
|  |  |  |

**LIST OF TABLES**

|  |  |  |
| --- | --- | --- |
| **TABLE NO.** | **TITLE** | **PAGE.NO** |
| Table 1.1 | Types of HTTP Methods | 8 |
| Table 1.2 | HTTP Headers | 9 |
| Table 1.3 | HTTP Status codes | 11 |

**LIST OF ABBREVIATIONS**

|  |  |
| --- | --- |
| HTTP | Hyper Text Transfer Protocol |
| HRS | HTTP Request Smuggling |
| OWASP  CL | Open Web Application Security Project  Content - Length |
| H2.CL | HTTP/2 Content Length |
| URI | Uniform Resource Identifier |
| TLS | Transport Layer Security |
| SSL | Secure Sockets Layer |
| LAN | Local Area Network |
| URL | Uniform Resource Locater |
| XSS | Cross Site Scripting |
| WAF | Web Application Firewall |

**CHAPTER 1**

**INTRODUCTION**

**1.1 GENERAL**

Hypertext Transfer Protocol (HTTP) is a widely used protocol for transmitting data over the internet and is a critical component of web security. Despite the security measures implemented in HTTP, it is still vulnerable to attacks. One of the vulnerabilities that has been exploited by attackers is the desynchronization of HTTP requests and responses, commonly known as HTTP desync attacks.

HTTP Desync attacks have become more common in recent years, and they can cause significant harm to web applications and users. These attacks involve manipulating the HTTP requests and responses in a way that causes the server to process them differently, leading to potential security breaches. These attacks are challenging to detect and prevent, making them a significant threat to web security.

The primary objective of the project is to provide a comprehensive understanding of HTTP desync attacks, encompassing their types, execution methods, and potential impact on web security. The vulnerabilities in the HTTP protocol that render it susceptible to desync attacks will be examined, alongside a thorough review of existing research and mitigation solutions.

The importance of securing HTTP communication and the need for effective countermeasures to prevent HTTP desync attacks are highlighted in the project. To develop effective solutions for mitigating the impact of desync attacks on web security, it is crucial to understand the techniques used in such attacks and the vulnerabilities they exploit.

**1.1.1 OVERVIEW**

The project aims to explore the lesser-known vulnerability of HTTP Request Smuggling (HRS) that poses a significant threat to web servers and proxies. The vulnerability mentioned above is often challenging to detect and mitigate, as it relies on subtle variations in the HTTP requests, making it difficult to spot. The overview of the project discusses the growing importance of cybersecurity due to increased digitization of activities and reliance on the internet, which exposes personal information to cyber threats. The Open Web Application Security Project (OWASP) has identified the top 10 web application security risks, but there are other vulnerabilities that exist, including HTTP desync attacks, which exploit the way web servers and proxies handle requests.

The project focuses on the HTTP Desync Attack technique known as H2.CL (HTTP/2 Content-Length), which exploits a vulnerability in the handling of Content-Length headers in HTTP/2. It highlights the significance of maintaining up-to-date security updates and implementing best practices to safeguard against HTTP Desync Attacks. The project aims to establish a solid understanding of HTTP/2, enabling more efficient utilization of network resources. By specifically examining the H2.CL technique, the project delves into its potential impact on web applications and provides recommendations for detecting and preventing such attacks, emphasizing the importance of thorough testing and proactive security measures.

**1.1.2 DESCRIPTION**

The project presents a detailed analysis of the H2.CL technique, an HTTP desync attack that exploits a vulnerability in the handling of Content-Length headers in HTTP/2. The analysis aims to understand the execution of the technique and its potential impact on susceptible web applications. The project discusses various methods for detecting and preventing H2.CL attacks, including upgrading web server and proxy software, implementing strict HTTP request parsing rules, and utilizing specialized security tools for monitoring and analyzing web traffic.

With a specific focus on H2.CL, the project aims to provide web developers and security professionals with a comprehensive understanding of the associated risks and actionable recommendations to enhance the security of their web applications. The goal is to raise awareness regarding the dangers posed by HTTP desync attacks and promote the adoption of improved security practices to mitigate future attacks.

**1.1.3 OBJECTIVE**

The objective of the project is to evaluate the effectiveness of different countermeasures for detecting and preventing H2.CL attacks. Practical experimentation will be conducted to assess the strengths and limitations of various approaches, such as upgrading server software, implementing stricter parsing rules, and utilizing specialized security tools. Additionally, the impact of HTTP/2 CL attacks on web applications will be investigated, along with the feasibility of different attack scenarios. The ultimate goal of the project is to provide recommendations for best practices to prevent and mitigate the risks associated with H2.CL attacks. The project aims to enhance the security posture of web applications and safeguard against potential data breaches or other security incidents.

**1.2 ABOUT THE PROJECT**

The project presents the results of a research project that aimed to investigate the security risks associated with the H2.CL technique, a type of HTTP desync attack that exploits a vulnerability in the handling of Content-Length headers in HTTP/2. The project involved conducting a thorough analysis of the technique, including its underlying principles, potential consequences for web applications, and available methods for detection and prevention.

The project included a review of relevant literature on the topic, as well as experimentation with different techniques for carrying out H2.CL attacks and testing the effectiveness of different countermeasures. In addition, the project involved collaboration with web developers and security professionals to gain insight into the practical implications of the technique and to gather feedback on proposed recommendations for mitigating the risk of HTTP desync attacks.

Overall, the project aimed to provide a comprehensive overview of the HTTP/2 CL technique and its potential impact on web application security. It offered practical recommendations for enhancing the security posture of web applications in the presence of vulnerability. By shedding light on the often-overlooked aspect of web application security, the project contributes to the development of more effective and robust security practices in the field.

**1.3 HTTP – BASICS**

HTTP is a protocol used for transmitting data over the internet. It is the foundation of data communication on the World Wide Web. HTTP operates on a client-server model, where a client sends a request to a server, and the server responds with the requested information.

The HTTP request consists of a request line, headers, and a message body (optional). The request line contains the request method (e.g. GET, POST, PUT, DELETE), the URI (Uniform Resource Identifier), and the HTTP version. The headers provide additional information about the request, such as the content type, the language, and the accepted encoding. The message body contains the data being sent with the request, such as form data or a file upload.

The HTTP response, on the other hand, consists of a status line, headers, and a message body (optional). The status line contains the HTTP version, the status code, and the status message. The headers provide additional information about the response, such as the content type, the encoding, and the cache control. The message body contains the data being sent with the response, such as the requested web page or a file. HTTP also supports several other features, such as caching, authentication, and compression. Overall, HTTP is a foundational protocol for the World Wide Web and plays a critical role in enabling communication between clients and servers.

**1.3.1 HTTP/1.0**

HTTP/1.0 was the first version of the HTTP protocol to be widely used on the internet. It was published in 1996 and was designed to allow for the transmission of simple text-based documents, such as HTML pages. One of the advantages of HTTP/1.0 is that it is a simple and straightforward protocol, which makes it easy to implement and understand. Additionally, it has low overhead and is suitable for low-bandwidth connections. However, HTTP/1.0 also has several disadvantages. For example, it is a connection-oriented protocol, which means that each client request requires a separate TCP connection to the server, resulting in high latency and increased server load. Additionally, HTTP/1.0 headers are limited in scope and functionality compared to later versions of the protocol, which can limit the ability to transmit more complex data.

**1.3.2 HTTP/1.1**

HTTP/1.1 is a version of the HTTP protocol that was introduced in 1999 as an upgrade to HTTP/1.0. It was designed to address some of the limitations of the earlier protocol and to support more efficient communication between clients and servers. One of the main advantages of HTTP/1.1 is its support for persistent connections, which allows multiple requests to be sent over a single connection, reducing latency and server load. Another advantage is its support for chunked encoding, which allows large data transfers to be broken up into smaller chunks and sent over the same connection. Additionally, HTTP/1.1 supports pipelining, which allows multiple requests to be sent without waiting for a response, further reducing latency.

However, there are also some disadvantages to HTTP/1.1. For example, it is still a text-based protocol, which can result in large message sizes and slow transmission speeds. Additionally, it can be vulnerable to security attacks, such as slowloris attacks, due to its connection reuse mechanism. Furthermore, HTTP/1.1 lacks support for server push, which means that the server cannot proactively send data to the client without a prior request, which can limit the ability to deliver real-time updates or streaming data.

Overall, HTTP/1.1 has been widely adopted and is still in use today, but its limitations have led to the development of newer versions of the protocol, such as HTTP/2 and HTTP/3, which aim to address some of these issues and provide even more efficient and secure communication between clients and servers.

**1.3.3 HTTP/2**

HTTP/2 is a major revision of the HTTP protocol, which was released in 2015. It was designed to address some of the limitations of HTTP/1.1 and to provide a more efficient and secure way to communicate between clients and servers. One of the main advantages of HTTP/2 is its support for multiplexing, which allows multiple requests and responses to be sent over a single connection, reducing latency and improving performance. Additionally, HTTP/2 uses binary framing, which reduces message size and simplifies parsing, leading to faster transmission speeds.

Another advantage of HTTP/2 is its support for server push, which allows the server to proactively send data to the client without a prior request, improving the ability to deliver real-time updates or streaming data. Additionally, HTTP/2 provides enhanced security features, such as support for encrypted connections using TLS/SSL and the ability to prioritize requests and responses, allowing for more efficient use of network resources.

However, there are also some disadvantages to HTTP/2. For example, it can be more complex to implement and debug compared to HTTP/1.1, and some older web servers and browsers may not support it. Additionally, HTTP/2 can be vulnerable to certain types of attacks, such as Slow Read attacks, which exploit the way the protocol handles flow control.

Overall, HTTP/2 represents a significant improvement over HTTP/1.1 and has been widely adopted by web developers and server administrators. It provides a more efficient and secure way to communicate between clients and servers, and is expected to continue to evolve and improve over time.

Having a basic understanding of how HTTP requests are sent to the server is crucial for a precise comprehension. An example of a simple HTTP request is provided in Figure 1.1 in which the request is sent to abc.xyz using HTTP/2 protocol.

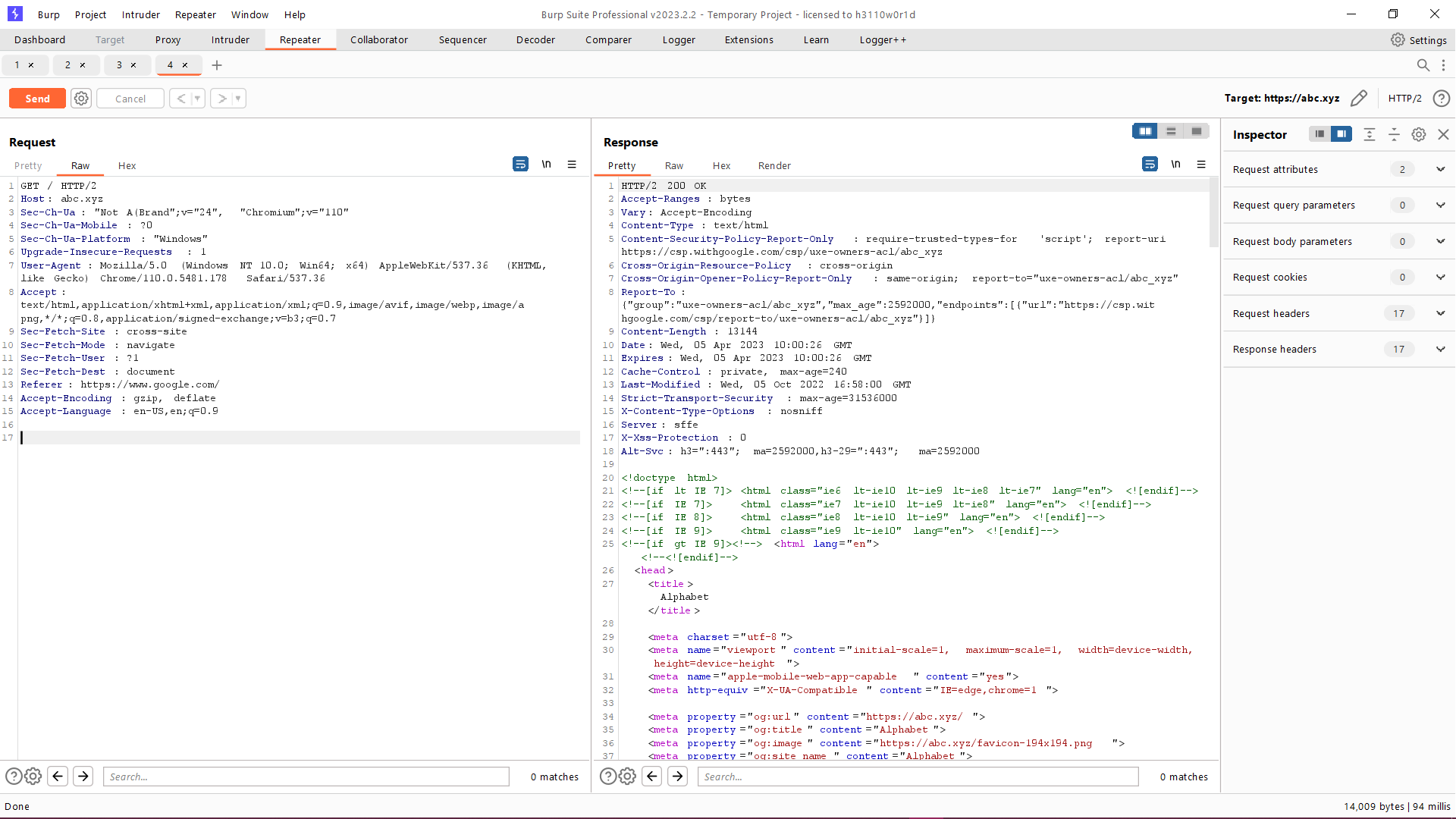


Figure 1.1 – Sending a request to [www.abc.xyz](http://www.abc.xyz)

In Figure 1.2, the corresponding HTTP response received from abc.xyz is displayed.

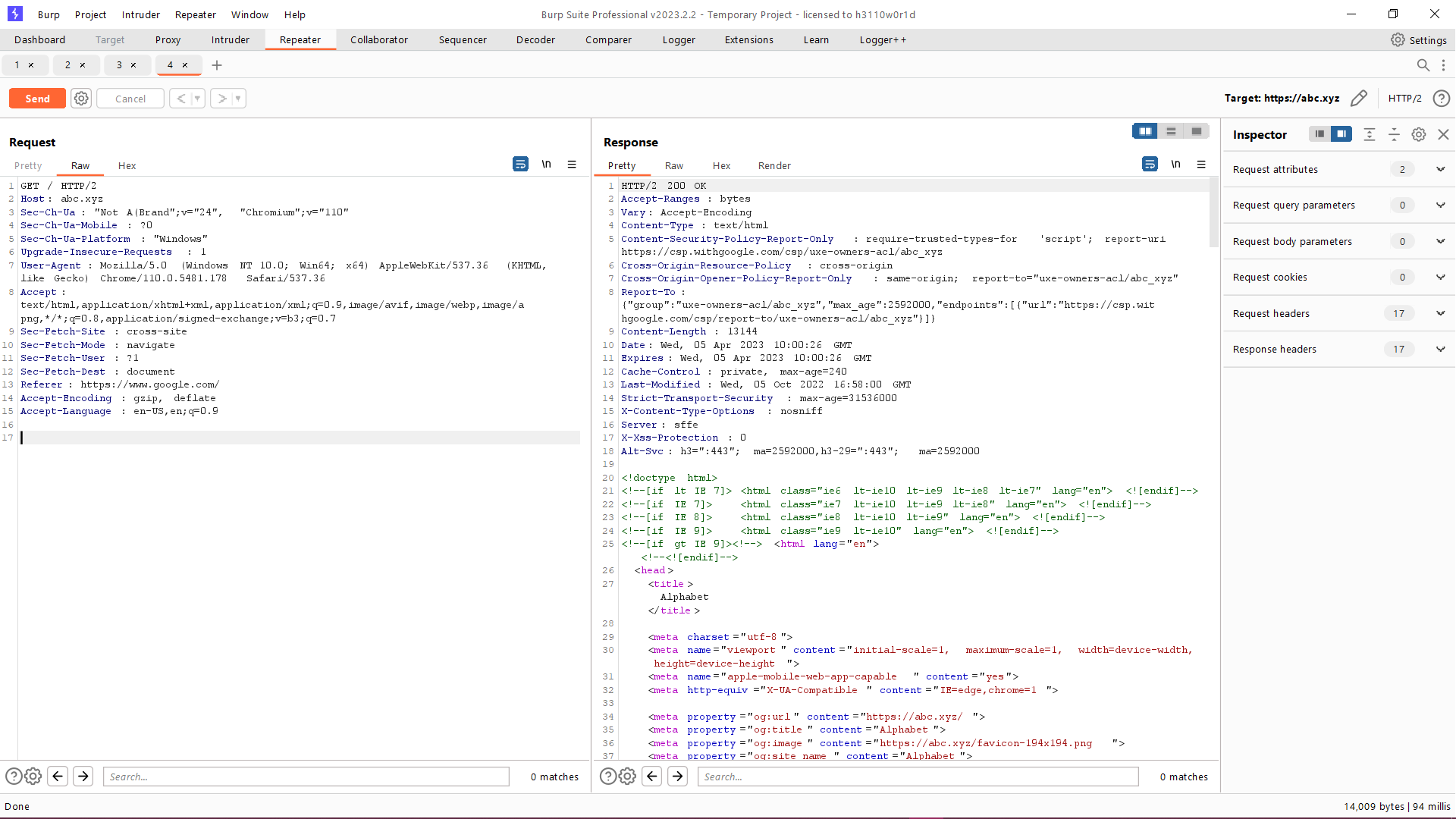


Figure 1.2 – Respective response for above sent request

**1.4 HTTP Request:**

When a client makes an HTTP request to www.abc.xyz, it typically includes several components, including:

1. The request method: It specifies the HTTP method used for the request, such as GET, POST, or PUT.

2. The request URI: It specifies the resource being requested, such as a web page, image, or other file.

3. Request headers: These are optional components that provide additional information about the request, such as the client's user agent, preferred language, and authentication credentials.

4. Request body: It is an optional component that may contain additional data, such as form data or JSON payloads.

**1.5 HTTP Response:**

When the server receives an HTTP request from a client, it generates an HTTP response that typically includes the following components:

1. Response status: It is a three-digit code that indicates the status of the response, such as 200 OK, 404 Not Found, or 500 Internal Server Error.

2. Response headers: These provide additional information about the response, such as the content type, length, and encoding.

3. Response body: It contains the actual content of the response, such as HTML, JSON, or an image file.

Overall, HTTP requests and responses are essential for web communication, allowing clients and servers to exchange data and resources in a standardized and efficient manner. By understanding these components, developers and server administrators can optimize their systems for maximum performance and security.

Figure 1.3 displays an HTTP/1.1 request sent to www.twitter.com, while Figure 1.4 shows an HTTP/2 request sent to the same website. The differences between the two requests can be clearly seen in the figures.

In the HTTP/1.1 request (Figure 1.3), the request method used was a GET request, and the URI specified the resource being requested. The request included several headers, such as the user-agent and accept-language headers. The response status code was 200 OK, indicating that the request was successful.

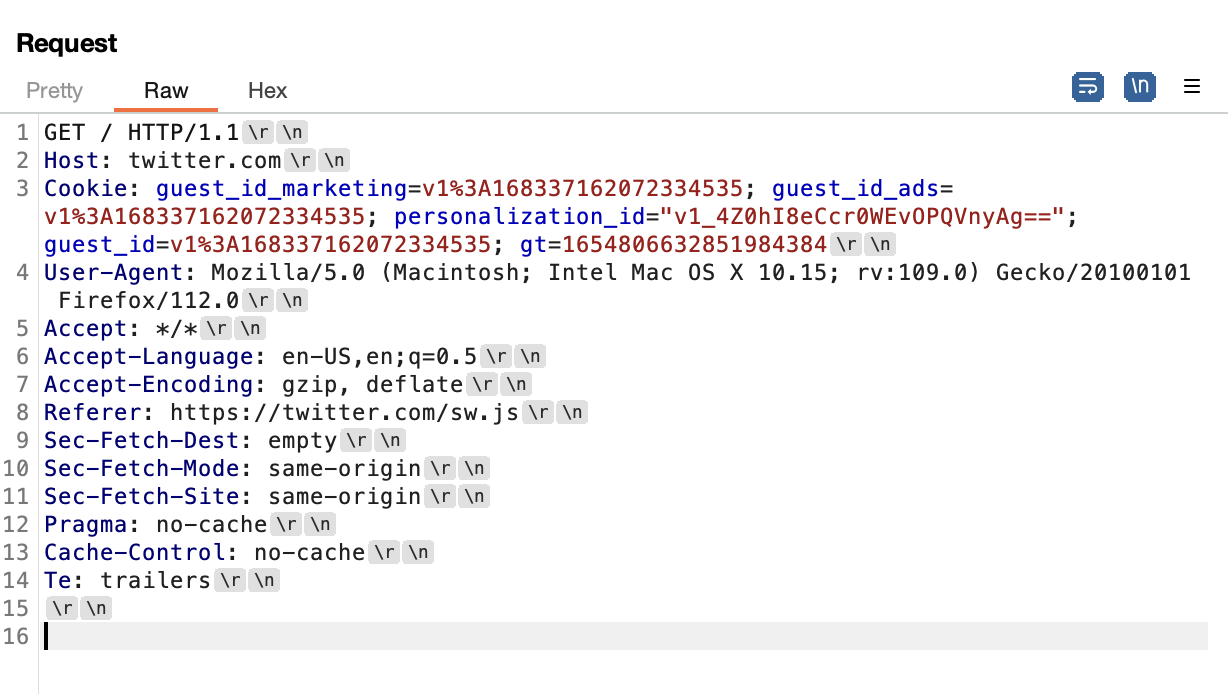


Figure 1.3 – Twitter HTTP/1.1 request

In contrast, the HTTP/2 request in Figure 1.4 uses a different format that includes binary framing, multiplexing, and header compression. It enables more efficient and faster communication between the client and server, resulting in improved performance and reduced latency.



Figure 1.4 – Twitter HTTP/2 request

The use of HTTP/2 can also result in other performance improvements, such as server push, which allows the server to push resources to the client without waiting for a request. Additionally, HTTP/2 supports stream prioritization, which enables the client to specify the priority of each resource request.

**1.6 HTTP METHODS**

The table 1.1 provides an overview of the different HTTP methods and their corresponding descriptions.

**Table 1.1 Types of HTTP Methods**

|  |  |
| --- | --- |
| **Method** | **Description** |
| GET | Requests a representation of the specified resource |
| POST | Submits an entity to be processed to the specified resource |
| PUT | Replaces all current representations of the specified resource with the request payload |
| DELETE | Deletes the specified resource |
| CONNECT | Establishes a network connection to a resource |
| HEAD | Requests a response identical to a GET request, but without the response body |
| OPTIONS | Describes the communication options for the specified resource |
| PATCH | Applies partial modifications to a resource |
| TRACE | Performs a message loop-back test along the path to the target resource |

Table 1.1 summarizes the different types of HTTP methods that can be used to interact with web resources. Understanding these methods is crucial for developing and securing web applications.

**1.6.1 HTTP HEADERS**

HTTP headers play a critical role in facilitating effective communication between clients and servers. They contain vital metadata, including content type, language, and caching options. In the project, a comprehensive list of HTTP headers, along with their descriptions and compatible HTTP versions, will be provided. It will help deepen the understanding of how these headers function in practice.

The table provides a list of HTTP headers and their descriptions, along with the compatible HTTP versions. Understanding these headers is crucial for effective communication between clients and servers.

**Table 1.2 HTTP Headers**

|  |  |  |
| --- | --- | --- |
| **Header** | **Description** | **HTTP Version(s)** |
| Accept | Specifies the format of the response that the client is willing to accept | HTTP/1.1, HTTP/2 |
| Content-Type | Specifies the format of the data in the request or response body | HTTP/1.1, HTTP/2 |
| User-Agent | Identifies the client making the request | HTTP/1.1, HTTP/2 |
| Referer | Specifies the URL of the web page that linked to the current page | HTTP/1.1, HTTP/2 |
| Authorization | Sends credentials to the server for authentication | HTTP/1.1, HTTP/2 |
| Cache-Control | Specifies caching options for the response | HTTP/1.1, HTTP/2 |
| Connection | Specifies whether to keep the connection open or close it after the request is completed | HTTP/1.1 |
| Content-Length | Specifies the length of the data in the request or response body | HTTP/1.1, HTTP/2 |
| Cookie | Sends cookies to the server | HTTP/1.1, HTTP/2 |
| Host | Specifies the hostname of the server that the client wants to connect to | HTTP/1.1, HTTP/2 |
| Upgrade | Specifies the protocol versions being used for the connection, typically used for HTTP/2 negotiation | HTTP/1.1 |
| Transfer-Encoding | Specifies the form of encoding used to safely transfer the message body to the user. | HTTP/1.1 |
| TE | Specifies the transfer encodings that are acceptable for the response | HTTP/1.1 |
| Accept-Encoding | Specifies the encodings that the client can handle for the response | HTTP/1.1 |
| If-Modified-Since | Allows conditional GET requests based on the modification time of the requested resource | HTTP/1.1 |
| Range | Allows clients to request specific ranges of the response entity, such as bytes 100-200 | HTTP/1.1 |
| Accept-Language | Specifies the natural language(s) of the response | HTTP/1.1 |
| DNT | Requests that the server does not track the user's browsing behavior | HTTP/1.1 |
| Origin | Specifies the origin of the request, typically used for Cross-Origin Resource Sharing (CORS) | HTTP/1.1 |
| Sec-Fetch-Site | Specifies the site that initiated the fetch, typically used for Cross-Site Request Forgery (CSRF) | HTTP/1.1 |
| Sec-Fetch-Mode | Specifies the mode of the fetch request, such as "cors" or "navigate" | HTTP/1.1 |
| Sec-Fetch-Dest | Specifies the destination of the fetched resource, such as "document" or "script" | HTTP/1.1 |
| Sec-WebSocket-Version | Specifies the WebSocket protocol version being used | HTTP/1.1 |
| Sec-WebSocket-Key | Used in the WebSocket handshake to generate the Sec-WebSocket-Accept response header | HTTP/1.1 |
| Upgrade-Insecure-Requests | Requests that the server upgrade to HTTPS before fulfilling the request | HTTP/1.1 |
| Alt-Svc | Specifies alternative services available for HTTP/1.1 clients to use instead of the current server | HTTP/1.1 |
| :authority | Specifies the authority part of the request URI, typically used in HTTP/2 | HTTP/2 |
| :method | Specifies the HTTP method of the request, such as GET or POST, typically used in HTTP/2 | HTTP/2 |
| :path | Specifies the path part of the request URI, typically used in HTTP/2 | HTTP/2 |
| :scheme | Specifies the scheme part of the request URI, typically used in HTTP/2 | HTTP/2 |
| :status | Specifies the status code and reason phrase of the response, typically used in HTTP/2 | HTTP/2 |

Table 1.2 provides a comprehensive list of HTTP headers commonly used for communication between clients and servers. It is important to understand the function of each header and its compatible HTTP version to ensure effective communication and optimize web performance.

**1.6.2 COMMON HTTP STATUS CODES AND DESCRIPTIONS**

HTTP status codes are important indicators of the communication between clients and servers. They help developers identify errors, optimize web performance, and ensure secure communication. The Table 1.3 presents a list of common HTTP status codes, with their names and descriptions.

**Table 1.3 HTTP Status codes**

|  |  |  |
| --- | --- | --- |
| **Status Code** | **Name** | **Description** |
| 100 | Continue | The server has received the initial part of the request and the client should proceed to send the remainder of the request. |
| 200 | OK | The requested resource was found on the server and is being returned to the client. |
| 301 | Moved Permanently | The requested resource has been permanently moved to a new URL. |
| 302 | Found | The requested resource has been temporarily moved to a new URL. |
| 400 | Bad Request | The server cannot understand the client's request, typically due to malformed syntax or missing parameters. |
| 403 | Forbidden | The client does not have permission to access the requested resource. |
| 404 | Not Found | The requested resource was not found on the server. This can happen if the URL is misspelled or if the resource has been deleted. |
| 405 | Method Not Allowed | The client uses an HTTP method that is not supported by the requested resource. |
| 429 | Too Many Requests | The client has sent too many requests in a given amount of time, and the server is temporarily unable to handle any more requests. |
| 444 | No Response | The server has not returned a response, typically due to connection problems. |
| 500 | Internal Server Error | An error occurred on the server while processing the request. This can happen due to a variety of reasons, such as a misconfigured server or a programming error in the server-side code. |
| 503 | Service Unavailable | The server is temporarily unable to handle the request, typically due to maintenance or overload. |

**1.7 HOW A SERVER HANDLES REQUEST AND RESPONSE**

Imagine you're a librarian in a library, and your job is to help people find books they're looking for. You have a lot of books on your shelves, each with a unique identifier, such as a call number. When someone comes to you and asks for a specific book, you check the shelves to see if it's available. If you find it, you take it off the shelf and give it to the person. If you don't find it, you tell the person that the book is not available.

In the analogy, you are the server, the books are the resources that the clients want to access, and the people asking for the books are the clients making requests. Just like the books in a library have unique call numbers, the resources on a server have unique identifiers, such as URLs. When a client wants to access a resource on a server, it sends a request to the server, just like a person asking you for a book.

The request typically contains the URL of the resource the client wants to access and other information, such as the type of request (e.g., GET, POST, PUT, DELETE), headers, and any data that the server needs to process the request. The request is sent over a network, which can be a local area network (LAN) or the internet.

When the server receives the request, it processes it to determine what the client wants and whether it has the requested resource. Just like you would check the shelves to see if a book is available, the server looks for the requested resource in its storage, which can be a hard disk or a memory.

If the server finds the requested resource, it sends a response back to the client, just like you would give a book to a person who asked for it. The response typically contains the requested resource and other information, such as the status code (e.g. 200 OK, 404 Not Found), headers, and any data that the client needs to process the response.

If the server doesn't have the requested resource, it sends a response back to the client saying that the resource is not available, just like you would tell a person that the book is not available. The response typically contains a status code indicating that the resource was not found, such as 404 Not Found.

Handling requests and responses involves a lot of steps, and servers use various protocols to communicate with clients over a network. One of the most common protocols is the Hypertext Transfer Protocol (HTTP), which is used for web applications.

When a client sends an HTTP request to a server, the request contains a method, such as GET, POST, PUT, or DELETE, that specifies the type of request. The request also contains a URL that identifies the resource the client wants to access, as well as headers that provide additional information, such as the type of data the client expects to receive.

The server processes the request and sends back an HTTP response, which contains a status code that indicates whether the request was successful or not, as well as headers and any data that the client needs to process the response. The data can be in various formats, such as HTML, JSON, XML, or plain text.

Handling requests and responses also involves security considerations, such as authentication and authorization. Servers use various methods to ensure that only authorized clients can access certain resources and perform certain actions. For example, a server might require clients to provide a username and password to access certain resources or perform certain actions.

In addition, servers use various techniques to handle high volumes of requests, such as load balancing and caching. Load balancing involves distributing incoming requests among multiple servers to improve performance and availability. Caching involves storing frequently accessed resources in memory or on disk to reduce the time it takes to serve those resources to clients.

Another important aspect of handling requests and responses is error handling. Servers need to be able to handle errors gracefully, such as when a resource is not found, when the server is overloaded, or when the client sends an invalid request. Error handling can involve sending an appropriate error response to the client, logging the error for troubleshooting, or taking other actions to recover from the error.

Overall, handling requests and responses is a complex and critical part of server-side programming. It involves various protocols, security considerations, performance optimizations, and error handling techniques. Good server-side programming practices can help ensure that servers are secure, reliable, and efficient, and can provide a good user experience for clients accessing the resources on the server.

**1.8 CACHING**

Caching is a common technique used in server-side programming to enhance performance and efficiency. It involves temporarily storing frequently accessed resources in memory or on disk, which enables them to be retrieved quickly and efficiently. It helps to reduce the time it takes to serve these resources to clients, which in turn improves the overall performance and user experience of a server. The benefits and best practices of caching will be explored in more detail in the following section.

There are two primary types of caching available:

* Server-side caching, and,
* Client-side caching.

**1.8.1 SERVER-SIDE CACHING**

Server-side caching is a technique that temporarily stores web files and data on the server to reduce its load and latency, resulting in improved performance and faster loading times for users. When a user requests a web page, the website retrieves the necessary data from the server, generates the webpage, and displays it to the user. Once the response has been sent back to the user, the server creates a copy of the webpage and stores it as a cache.

The advantage of the mentioned technique is that the next time the user visits the website, the server can quickly serve the already saved copy of the webpage, bypassing the time-consuming process of generating it again from scratch. It not only improves the user experience but also reduces the workload on the server, resulting in improved efficiency and reduced response time. By leveraging server-side caching, website owners can provide a faster and more responsive experience to their users, ultimately leading to higher engagement and satisfaction.

The Figure 1.5 shows how a client request is first sent to the caching server, which checks if the requested data is already stored in its cache. If the data is present, it is quickly served back to the client. Otherwise, the caching server forwards the request to the web server, which generates the response and sends it back to the client. The caching server saves a copy of the response in its cache for future requests as shown in Figure given below.

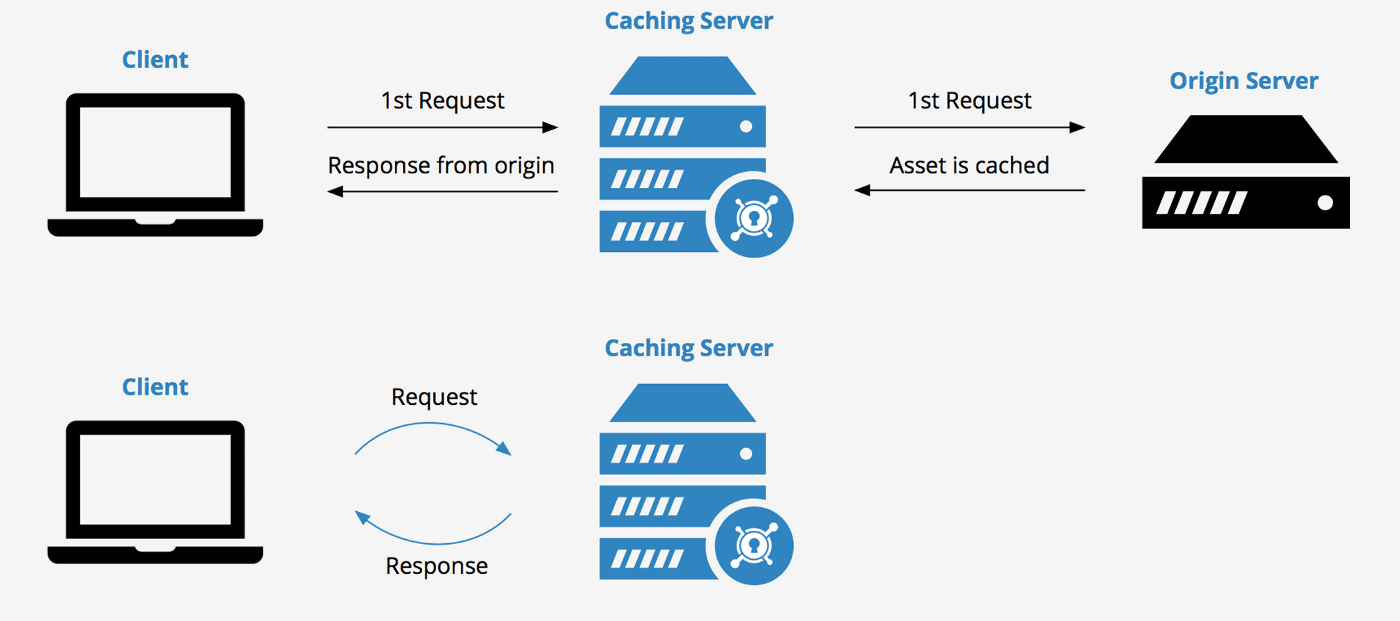


Figure 1.5: Illustration of server-side caching process.

**1.8.2 CLIENT-SIDE CACHING**

Client-side caching, sometimes referred to as browser caching, is a method of storing website data in the memory of the user's browser. When a user visits a website, the browser checks if it already has a cached version of the requested page (as shown in the Figure 1.6). If it does, the browser retrieves the page from the cache folder instead of requesting it from the server. It reduces the amount of data that needs to be transferred between the client and server, resulting in faster website loading times. The technical process of client-side caching involves sending a request from the client to the server, and then saving the webpage in the cache folder on the client's device.

Illustration of how client-side caching works - Source: Medium


Figure 1.6: Illustration of client-side caching process.

Furthermore, client-side caching can also save server resources, as it reduces the number of requests made to the server. It can be particularly useful in situations where the website has a high volume of traffic or where the server's resources are limited.

It's worth noting that client-side caching can be controlled by website developers using HTTP headers, which provide instructions to the browser on how long to store the cached content. If a developer sets a long cache time, it can result in stale content being displayed to users. Conversely, setting a short cache time can increase the number of requests made to the server, which can lead to slower load times for users.

**1.9** **UNDERSTANDING THE COMPONENTS OF A WEB SERVER: AN OVERVIEW OF FRONT-END AND BACK-END DEVELOPMENT**

A website or web application has two main components: the front-end and the back-end. The front-end is the user interface that users interact with directly, while the back-end manages data storage, processing, and retrieval. Both components work together to provide a seamless web experience for users.

HTTP/2 is a protocol that improves web communication and offers several optimizations to enhance performance. One of the significant improvements is the ability for the front-end to multiplex multiple requests and responses over a single TCP connection. It reduces latency and is particularly useful for web applications that require real-time updates. Additionally, HTTP/2 provides header compression that minimizes data transmission between the front-end and back-end of a server, leading to faster transmission and processing of requests. The feature is especially helpful for large websites with many resources.

By leveraging HTTP/2, both the front-end and back-end of a server can improve performance, reduce latency, and provide a better web experience for users.

When developing a website or web application, it is essential to consider both the front-end and back-end components to ensure a seamless and optimized user experience. The front-end handles the presentation layer, while the back-end manages the business logic and data storage.

HTTP/2 is a significant improvement over the previous HTTP/1.x protocols, offering several optimizations that can benefit both the front-end and back-end of a server. By implementing HTTP/2, websites and web applications can enhance performance, reduce latency, and provide real-time updates to users. The benefits of HTTP/2 make it a valuable addition to any web development project.

Figure 1.7 depicts the basic architecture of a server with a front-end and back-end. The front-end handles user requests, manages network connections, and pipelines requests for processing in the back-end. The back-end processes requests, generates dynamic content, and manages access to resources such as databases. Users send requests to the front-end, which manages incoming requests from clients and controls network connections.

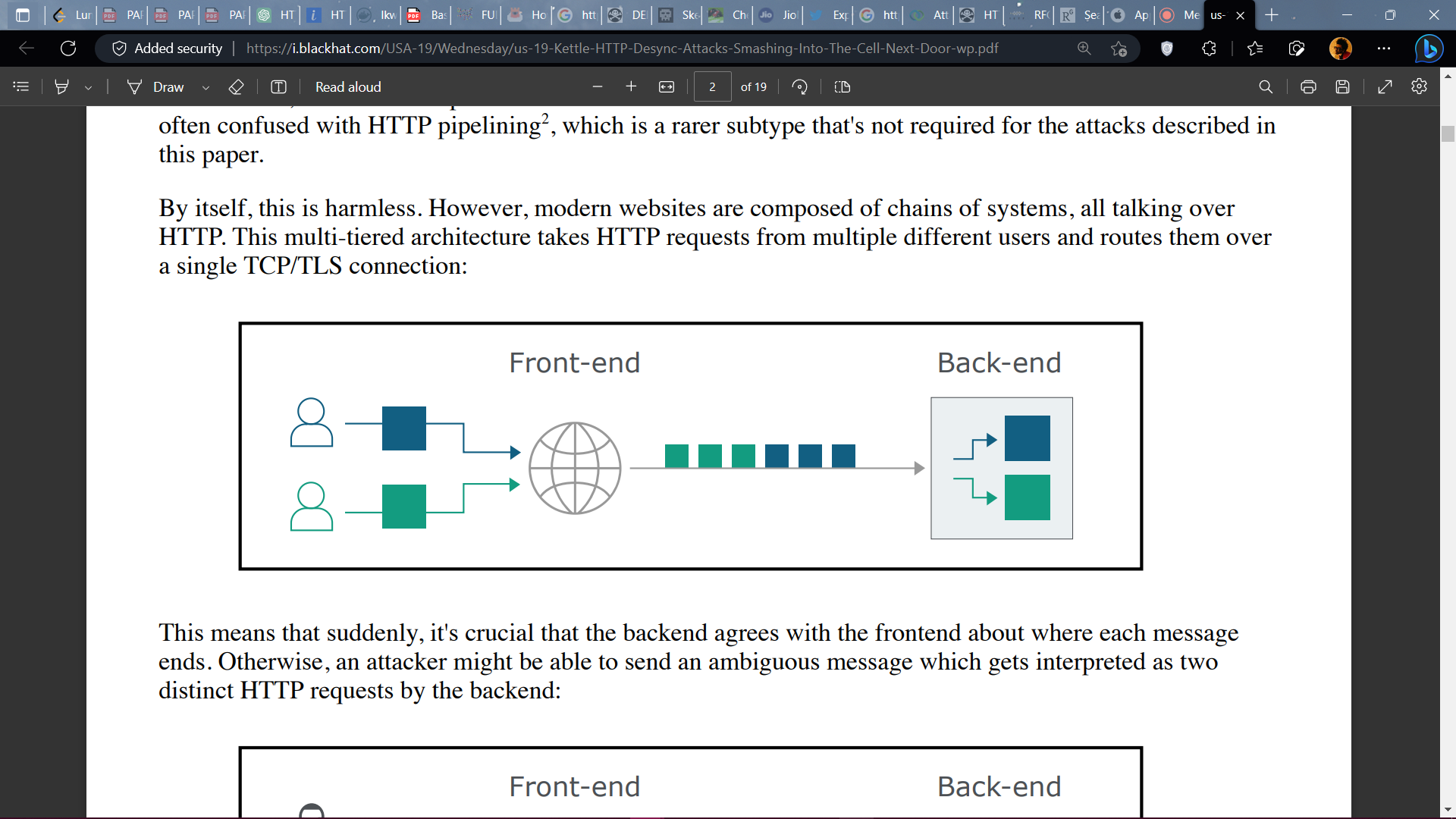


Figure 1.7 – Showing the working of a server

The front-end takes the user requests and pipelines them for processing in the back-end. Pipeline processing means that the front-end sends multiple requests to the back-end over a single connection in a queued manner, so that they are processed by the back-end in the order they were received. The technique is particularly useful for web applications that require multiple resources or real-time updates.

The back-end, on the other hand, is responsible for processing these requests, producing dynamic content, and accessing resources such as databases. Once the back-end has processed the requests, it sends responses back to the front-end, which in turn sends them back to the users.

**1.10 HTTP DESYNC ATTACK AND TYPES**

Web application security is a crucial aspect of internet security, and various types of weaknesses and vulnerabilities exist in web applications. The OWASP Top 10 Web Application Security Risks is a project dedicated to monitoring and addressing the most common security issues in web applications, such as SQL injection and cross-site scripting (XSS). However, several lesser-known vulnerabilities exist, including HTTP request smuggling (HRS).

HTTP request smuggling attacks are a type of HTTP desync attack that exploit the differences in how web servers and proxies interpret HTTP requests. Attackers send specially crafted HTTP requests that can be interpreted differently by different components of the web server or proxy, causing the server to behave in unexpected ways. These attacks can lead to various security issues, including the bypassing of security controls such as firewalls, intrusion detection and prevention systems, and web application firewalls. Attackers can also use HTTP desync attacks to inject malicious content into legitimate HTTP traffic, steal sensitive data, or even take control of vulnerable web servers.

Detecting and mitigating HTTP desync attacks can be challenging, as they often rely on subtle variations in the HTTP requests that can be difficult to detect. However, there are several techniques and tools available to help prevent and detect these attacks. Updating web server and proxy software to the latest version is a crucial step in preventing HTTP desync attacks, as newer versions often include fixes for known vulnerabilities. Implementing strict HTTP request parsing rules can also help prevent these attacks by ensuring that only valid HTTP requests are processed.

Specialized security tools can also be used to monitor and analyze web traffic for signs of HTTP desync attacks. These tools can detect and alert administrators to any unusual HTTP traffic patterns, allowing them to take action to mitigate the attack. Additionally, web application firewalls can be configured to detect and block HTTP desync attacks by examining the HTTP traffic for signs of these attacks.

HTTP desync attacks are a type of attack that involves sending a specially crafted HTTP request that can be interpreted differently by various components of a web server or proxy. Figure 1.8 illustrates the mechanism of an HTTP desync attack with two users, the attacker, and the user. The attacker sends a request that contains both a normal request and a malicious request smuggled inside it.

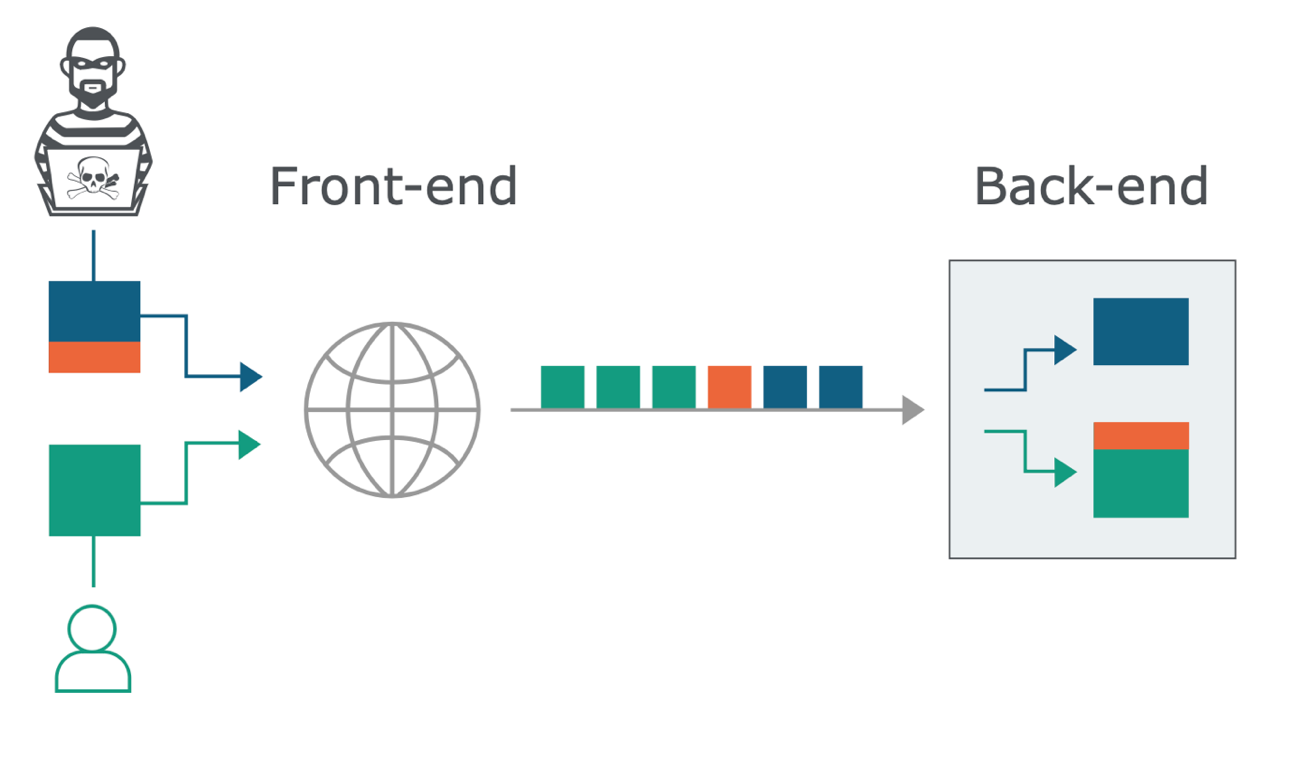


Figure 1.8 HTTP Desync Attack Mechanism

HTTP desync attacks can be carried out through multiple techniques, some of them includes:

* Request Smuggling: It is a type of HTTP desync attack that exploits the way web servers and proxies handle requests that have different content-length headers. Attackers can send two or more requests, where the second request is hidden or "smuggled" within the first request. The web server or proxy may interpret these requests as a single request, leading to various security issues.
* Response Splitting: It involves injecting a new line character or other special characters in the HTTP response headers. This can cause the web server or proxy to interpret the response as two separate responses, leading to various security issues.
* Chunked Encoding: It exploits the way web servers and proxies handle HTTP requests that use the "chunked" encoding transfer mechanism. Attackers can manipulate the chunked encoding size to cause the web server or proxy to interpret the request in unexpected ways.
* HTTP/2 Attacks: HTTP/2 is a newer version of the HTTP protocol that is designed to improve web performance. However, it also introduces new vulnerabilities that attackers can exploit. For example, attackers can use HTTP/2 to send multiple requests over a single TCP connection, leading to various security issues.
* Browser powered Desync attack: A Browser powered Desync attack is a sophisticated HTTP request smuggling attack that exploits the behavior of web browsers to manipulate the interpretation and processing of HTTP requests between the front-end and back-end servers. The attack involves sending a sequence of customized HTTP requests to the server, designed to trick the front-end and back-end servers into processing the requests in a way that violates the HTTP protocol standards. As a result, the attacker may gain unauthorized access, tamper with sensitive data, or execute unauthorized actions on the server. The term "browser powered" highlights the fact that these types of attack relies on the specific behavior of the browser, which may differ across various web applications and platforms. It makes it a challenging and complex attack to defend against.
* Hop-by-Hop method: The term "Hop-by-Hop" in HTTP pertains to certain headers that should only be used for a single connection and not be forwarded by proxy servers to the next hop. These headers include "Connection", "Keep-Alive", "Proxy-Authenticate", "Proxy-Authorization", "TE", "Trailer", and "Transfer-Encoding". Properly functioning proxy servers are expected to eliminate these headers from incoming requests and include them back to the response sent to the client, as stated in the HTTP specifications. If these headers are not removed, HTTP request smuggling attacks can occur, wherein an attacker can insert a hop-by-hop header to bypass security measures.
* Cache poisoning: Cache poisoning in HTTP request smuggling is an attack type that exploits vulnerabilities in caching systems to manipulate the caching behavior of servers and proxies, potentially resulting in the delivery of harmful content to clients or bypassing security controls. The attacker sends well-crafted requests that force the cache to store harmful data. When a legitimate client requests the same resource, the cache delivers the harmful data instead of the intended content. It can enable the attacker to steal sensitive information or perform unauthorized activities on behalf of the client. It is essential to detect and prevent cache poisoning attacks to ensure secure and reliable web communication.

These are just a few examples of the techniques that attackers can use it to exploit the vulnerability. As with any type of cyber-attack, attackers are constantly evolving their tactics, so it's essential to stay up-to-date with the latest security updates and implement best practices to protect against these attacks.

While there are various techniques and methods that can be used to launch HTTP desync attacks, it project focuses specifically on the “H2.CL attack” vector. Despite the existence of other attack types, we have chosen to concentrate solely on the particular method to provide a more in-depth analysis and understanding of its mechanisms and potential impact on web applications.

**1.10.1 H2.CL**

HTTP/2 request smuggling is a type of vulnerability that affects some servers handling HTTP/2 requests. H2.CL (HTTP/2 Content-Length) is one specific type of HTTP/2 request smuggling attack that exploits the vulnerability.

The H2.CL attack works by sending specially crafted HTTP/2 requests that contain a "Content-Length" header that is larger than the actual size of the request. It causes the server to treat the next request as a continuation of the previous one, leading to confusion about the boundaries between the requests. The confusion arises because the HTTP/2 protocol uses binary framing to encapsulate HTTP messages, and the "Content-Length" header is used in HTTP/1.1 to specify the length of the message body.

An attacker can exploit the confusion to perform various types of attacks. For example, an attacker can smuggle a malicious request past a web application firewall (WAF), which is designed to detect and block attacks. By bypassing the WAF, the attacker can execute a command or steal sensitive data from the server.

To carry out an H2.CL attack, an attacker must first identify a vulnerable server that supports HTTP/2 and is vulnerable to request smuggling. They can then craft a specially designed HTTP/2 request with a large "Content-Length" header to confuse the server and smuggle their malicious request past the WAF.

The H2.CL attack can have serious consequences and can lead to the compromise of sensitive data or the execution of malicious code on the target server. It is important for organizations to take steps to mitigate the vulnerability by applying patches and updates to vulnerable servers and monitoring for signs of an attack.

**1.11 FLOW DIAGRAM**

The Figure 1.9 outlines the proposed system workflow for detecting and preventing HTTP desync attacks in web servers.

Figure 1.9 - PROPOSED SYSTEM WORKFLOW – FLOWCHART

**I. DETECTION**

The first step in the workflow is the "Detection" stage, which is responsible for detecting any potential HTTP desync attacks. To accomplish it, a Python tool called Desyncron was developed. Desyncron is designed to use various techniques to detect HTTP desync attacks and does not require a proxy to check incoming requests. Desyncron is simple to use and can be installed and executed on any system with Python installed. To use it, developers need to provide the URL of the web application they want to test. Desyncron sends requests to the web application and analyzes the responses to detect any signs of HTTP desync attacks.

**II. CONFIRM**

In the second step of the proposed system workflow, the goal is to confirm the vulnerability. To achieve that, Turbo Intruder can be used.

Turbo Intruder is a powerful web application security testing tool that can be used to test for various types of vulnerabilities, including HTTP desync attacks. It is built on top of the Burp Suite, an integrated platform for performing security testing of web applications.

Turbo Intruder works by sending multiple HTTP requests simultaneously and analyzing the responses for any anomalies or inconsistencies. It allows it to quickly identify any potential HTTP desync issues and confirm their existence. By using Turbo Intruder in the confirmation step, the proposed system workflow can ensure that any potential HTTP desync attacks that were detected in the first step are indeed valid and require further action.

**III. EXPLORE**

If the attack is confirmed, the workflow moves to "Explore" step in the workflow which is a critical stage in understanding the attack and the vulnerabilities it is exploiting. In it, developers use various techniques to gain a deeper understanding of the attack and identify its root cause. These techniques may include the hop-by-hop method and frame-by-frame parsing in HTTP/2, among others. By method chaining these techniques, developers can analyze the attack more thoroughly and identify any inconsistencies or abnormalities.

The explore step also involves reverse engineering the attack or analyzing server logs to gather relevant information. Developers may need to examine each hop between the client and server to identify any inconsistencies. They may also need to analyze the sequencing of frames in the attack. By utilizing these techniques, developers can gain valuable insights into the attack and the vulnerabilities it is exploiting, allowing them to take appropriate action to prevent similar attacks in the future. Ultimately, the "Explore" step plays a crucial role in improving the overall security of web applications.

**IV. EXPLOIT**

The "Exploit" stage is the next step in the workflow after gaining an in-depth understanding of the HTTP desync attack in the "Detection" and "Explore" steps. In the current stage, developers aim to exploit the vulnerabilities identified earlier and recreate the attack under controlled conditions, utilizing the knowledge gained in the previous steps. The primary objective of it is to thoroughly understand the attack and determine its potential impact on the system. By replicating the attack, developers can assess the effectiveness of their countermeasures and identify any weaknesses that require attention. They can also evaluate the attack's impact under different scenarios and configurations to determine how the system responds. Moreover, the "Exploit" stage enables developers to evaluate the severity of the attack and prioritize their efforts accordingly, focusing on developing more robust countermeasures if the attack has the potential to cause significant damage.

**V. PREVENTION**

After gaining a comprehensive understanding of the HTTP desync attack and replicating it under controlled conditions in the "Exploit" stage, the next step in the workflow is "Prevention". In the prevention stage, the system implements measures to prevent future attacks.

One of the most critical measures in the prevention stage is to patch the vulnerabilities that have been exploited in the previous attack. Security patches or updates can be applied to web application components to eliminate the vulnerabilities that were identified in the previous stages. Additionally, any unnecessary components or functionalities that may pose a security risk can be removed. It can significantly reduce the attack surface and prevent attackers from exploiting any known vulnerabilities.

Another effective prevention measure is to implement web application firewalls (WAFs). WAFs monitor incoming and outgoing traffic to the web application and identify any malicious requests that may indicate an HTTP desync attack. They can block or filter out such requests before they reach the server, preventing the attack from being successful. By implementing a WAF, developers can add an additional layer of security to their web application and reduce the likelihood of HTTP desync attacks.

**VI. MAINTENANCE**

In conclusion, the maintenance stage of the workflow is crucial for ensuring that web applications are continually protected against potential threats. One of the most important aspects of maintenance is to keep the WAF up-to-date with the latest rules and techniques to address emerging threats. As new vulnerabilities are discovered and new attack techniques are developed, WAF rules must be updated to protect against those threats. A centralized GitHub repository provides a convenient location for users to access the latest updates and ruleset, allowing them to stay up-to-date with the most recent security measures. By regularly updating the WAF and following best practices for web application security, organizations can ensure that their web applications remain secure and protected against malicious attacks.

**CHAPTER 2**

**LITERATURE SURVEY**

Min-Yi Chiu et al. [1] discuss the history and evolution of research on HTTP request smuggling attacks, including a significant breakthrough in 2015 and the introduction of a new attack technique in 2019. The paper identifies the need for an effective defense method and proposes a new approach based on Flask as a reverse proxy. This defense system ensures that a user's request complies with RFC standard and passes the regex before being sent to the back-end server, making it easy to implement and extend to other attack methods.

Chaim Linhart et al. [2] introduced the term HTTP Request Smuggling (HRS) in their 2005 paper, which provides a detailed explanation of how HRS exploits parsing discrepancies in HTTP entities during data flow between the user and the web server. By sending multiple crafted requests, HRS causes the entities to see different sets of requests, allowing a request to be smuggled to one device without the other being aware. This vulnerability can be exploited to launch attacks such as cross-site scripting, web cache poisoning, session hijacking, and bypassing web application firewall protection, leading to credential hijacking and other malicious outcomes.

B. Jabiyev et al. [3] conducted a systematic exploration of HTTP Request Smuggling (HRS) within a scientific framework in their paper "T-reqs: HTTP request smuggling with differential fuzzing," published in 2021. The authors presented the first such study of HRS in a proxy-origin configuration, which exploits discrepancies in HTTP processing between two servers, enabling hidden requests to be smuggled through the proxy. To test popular server/proxy/CDN technologies and identify pairs that result in processing discrepancies, the authors used a novel grammar-based differential fuzzer. The experiment revealed previously unknown methods for manipulating HTTP requests for exploitation and documented server pairs that are susceptible to HRS.

J. Kettle [4] emphasizes the severity of HTTP desync vulnerability in bypassing various security measures. To prevent these attacks, the author recommends using HTTP/2 for request authentication, and implementing input validation and sanitization in web applications. Additionally, Kettle has developed a Burp Suite extension named "Desync-Proxy" for detecting and exploiting HTTP desync vulnerabilities. The work done by Kettle has considerably contributed to the research and development of mitigation techniques to enhance web application security and raised awareness of this vulnerability.

R. Fielding and J. Reschke [5] describes the basics and specification of HTTP/1.1, which replaces RFC 2616 and RFC 2145 on HTTP versioning. This specification updates the use of CONNECT to establish a tunnel and defines the "https" Uniform Resource identifier (URI) scheme. HTTP presents a uniform interface to clients in information systems, independent of the types of resources provided, and it can be used in various contexts, allowing implementations to evolve independently over time.

Amit Klein [6] proposed a technique that exploits differences in the interpretation of non-standard HTTP requests by various HTTP devices to smuggle malicious requests to the server. The attacker manipulates the way HTTP tools split the stream into individual requests to execute the attack. Despite ongoing research in the field since this technique was invented in 2005, current open-source defense systems such as mod\_security's community rule-set are often rudimentary and ineffective, and the paper explores the need for more advanced defense mechanisms.

Mattias Grenfeldt and Asta Olofsson [7] conducted an empirical study on HTTP Request Smuggling (HRS) in open-source servers and proxies where they tested how web servers and proxies interpret the length of an HTTP request by compiling a corpus of requests that employ various HRS techniques and their variations through literature review. The study discovered a total of 17 vulnerable behaviors, and combining proxies with servers, two nearly complete and three complete attacks were feasible. Although each tested system had at least one behavior that contradicted the HTTP specification, not all of these behaviors enabled HRS. In conclusion, the study found that most proxies had strict parsing and rejected requests that might result in HRS, while servers were less strict.

S. A. Mirheidari et al. [8] explored the Web Cache Deception (WCD) technique that allows sensitive content to be stored in a web cache, making it easily accessible to anyone on the internet. Earlier studies focused on the leakage of personal information on authenticated websites but failed to recognize the full extent of the attack's consequences. The researchers proposed a new method for detecting WCD, involving page identicality checks and cache header heuristics, that can be applied to any website without requiring marker injection. The study identified 1188 vulnerable websites in the largest WCD experiment to date, conducted on the Alexa Top 10K. The researchers demonstrated through case studies that attacks on non-authenticated pages can have severe consequences.

H. Nguyen et al. proposed a new type of web cache poisoning attacks, named Cache Poisoned DoS (CPDoS), which exploits a vulnerability on the origin server to poison the cache and cause denial-of-service (DoS) to the target service. The authors conducted a broad study of 15 web caching solutions and discovered that one proxy cache product and five CDN services are vulnerable to CPDoS attacks, including solutions that cache high-value websites. A single request can lead to severe effects, such as rendering the victim's website unavailable in a large geographical area. The paper provides valuable insights for practitioners and researchers to develop countermeasures and secure distributed systems.

Tyson et al. [10] studied the manipulation of HTTP headers, which has become increasingly common. The authors collected data from thousands of networks to understand regional and network trends. The study found that 25% of the networks modify HTTP headers, and that there are distinct trends among different regions and network types. For example, weakly connected regions have good numbers of cache headers. Additionally, the paper provides an in-depth analysis of the types of manipulations and how they differ across regions.

M. A. Wazzan and M. H. Awadh's [11] proposed a method to improve web attack detection by identifying the significant factors that affect the detection accuracy. The authors surveyed the literature on web attack detection and identified the common factors that affect the accuracy of detection. These factors include the selection of features, the classification algorithm used, the dataset used for training and testing, and the evaluation metrics used to assess the accuracy of detection. The authors found that using a combination of both network-level and application-level features led to the best performance.

R. Leroy [12] provides a brief overview of HTTP smuggling and its various types, such as CL.TE and TE.CL smuggling, and their potential impact on web application security. He also discusses the challenges faced in detecting HTTP smuggling attacks and provides insights into the methods used by attackers to circumvent existing detection mechanisms. The post also includes a detailed analysis of several test cases to demonstrate how HTTP smuggling attacks can be carried out. It should be noted that the information presented is from 2015 and may not be fully up to date with current security trends and practices. It is essential to rely on more recent research and updates to stay current with the latest security threats and mitigation techniques.

Doyhenard [13] explains how attackers can exploit the HTTP/1.1 protocol to carry out HTTP response smuggling attacks, which involve manipulating the interaction between a web server and client in order to bypass security mechanisms and carry out attacks such as cross-site scripting (XSS) and cross-site request forgery (CSRF). The paper also presents a number of real-world examples of HTTP response smuggling attacks, demonstrating the practical implications of this type of attack. Overall, Doyhenard's paper provides a comprehensive overview of the HTTP response smuggling attack, highlighting its potential impact and the various ways in which it can be carried out. The paper remains relevant today as HTTP/1.1 is still widely used, and the techniques and principles presented in the paper continue to be relevant for understanding and mitigating this type of attack.

R. Fielding and J. Reschke [14] proposed a complete overview of the protocol, covering message syntax, request methods, status codes, and routing through intermediaries such as proxies and gateways. The authors offer detailed explanations of the workings of the protocol, making this RFC a valuable reference for those who need to comprehend the inner workings of HTTP/1.1. Overall, this document remains relevant today, as HTTP/1.1 continues to be widely used, and its principles and guidelines continue to inform the development and deployment of web applications and services.

Jon Postel [15] describes TCP as one of the primary protocols used for transmitting data across the internet, and this RFC document provides a detailed explanation of its design, operation, and various functions. The author discusses the fundamental aspects of TCP, including its connection management, data flow control, and error recovery mechanisms. The document also covers TCP's header fields and flags, and its use in ensuring efficient network operation. This document is an authoritative resource for network administrators and developers to understand TCP's features and functions for effective use of TCP-based applications and networks.

**CHAPTER 3**

**PROBLEM DEFINITION AND METHODOLOGIES**

**3.1 PROBLEM DEFINITON**

While the base paper on HTTP desync attacks provides a comprehensive analysis of the vulnerability in the HTTP/1.1 protocol that can be exploited to execute desync attacks, it does not address modern HTTP request smuggling techniques, including HTTP/2 desync attacks, downgrade attacks from HTTP/2 to HTTP/1.1, or browser-powered desync attacks.

These modern techniques pose a serious threat to web security, as they enable attackers to manipulate HTTP requests and responses, causing the server to process them in unexpected ways, leading to potential security breaches. For example, HTTP/2 desync attacks can manipulate HTTP/2 streams to cause the server to interpret them differently, bypassing security measures, and resulting in unauthorized access to sensitive data.

Given the growing prevalence of these attacks, it is essential to extend the base paper's analysis and examine modern HTTP request smuggling techniques and their potential impact on web security. The project aims to review the current state of research and existing solutions to mitigate these attacks and propose new approaches to address the limitations of the base paper.

**3.2 EXISTING SYSTEM**

The current approach used in the HTTP desync system is based on exploiting vulnerabilities in the HTTP/1.1 protocol to carry out the attack. The system uses open-source software to exploit the vulnerability, with the aim of detecting and preventing HTTP request smuggling attacks. However, it is limited as it only focuses on the HTTP/1.1 protocol, and may not be effective against HTTP/2 or other protocols. Additionally, as the system relies on open-source software, it may be vulnerable to other types of attacks or exploits targeting the same software components. Overall, while the existing system provides some level of protection against HTTP request smuggling attacks, it may not be comprehensive or effective enough in all scenarios.

**3.2.1 EXISTING SYSTEM TECHNIQUE**

In their system they used Flask as a reverse proxy to identify and prevent HTTP request smuggling. Existing approach entails ensuring that all user requests adhere to the RFC standard and pass through a regex check before being forwarded to the back-end server for inspection. By implementing the stated method, website operators can safeguard against these types of attacks without the need for complicated installation, operation, or configuration. Furthermore, the proposed defense technique is straight-forward to expand to include other forms of attacks.

**3.2.2 ISSUES IN EXISTING SYSTEM**

* It is difficult to detect and mitigate, especially for less experienced developers and web administrators.
* Additionally, the attack requires detailed knowledge of the target web server and its configuration, which can make it more challenging to execute successfully.
* Another issue is that the attack may not always work as expected, depending on the specific server configuration and the complexity of the attack payload. It can make it difficult to predict the effectiveness of the attack and may require multiple attempts to succeed.
* Finally, there is a risk that attempting the attack could lead to unintended consequences or cause unexpected errors on the server. It could result in service disruptions or other issues that may impact the availability and performance of the targeted system.

**3.3 PROPOSED SYSTEM**

* Python tool specifically designed to detect the HTTP Desync attack in websites. The tool automates the detection process by generating a multitude of requests, each employing different HTTP Desync techniques. By systematically varying the headers and parameters in these requests, we aim to uncover potential vulnerabilities in the website's HTTP processing and identify any instances of desynchronization between the client and server. The tool's ability to generate a large number of requests with different techniques significantly increases the chances of detecting subtle vulnerabilities that may go unnoticed through manual inspection alone.
* To enhance the accuracy and reliability of the detected vulnerabilities, we employ a manual confirmation process using established tools like Burpsuite, Turbo Intruder, and backend log analysis. Burpsuite allows manual examination and manipulation of HTTP requests and responses, enabling further investigation and validation of potential vulnerabilities identified by the Python tool. Turbo Intruder automates payload generation, facilitating targeted attacks to validate detected vulnerabilities. Backend log analysis cross-references detected anomalies with server-side behavior and log records, adding an extra layer of confirmation. The comprehensive manual confirmation process ensures the reliability and accuracy of the vulnerability detection performed by the Python tool.
* Taking the detection and confirmation process a step further, we delve into the exploration of vulnerability chaining to create a higher impact.
* During the process of downgrading from HTTP/2 to HTTP/1.1 protocol, the proposed system takes necessary measures to ensure that the request headers are valid and authorized. It achieves by analyzing and validating the request headers, and allowing only those headers that are considered valid and authorized as per the predefined rules set by the administrator. By implementing it, the system can prevent malicious attacks that may occur due to invalid or unauthorized request headers, thus enhancing the security and reliability of the overall system.

**3.3.1 ADVANTAGES OF PROPOSED SYSTEM**

* The proposed system implements WAF rules that ensure only secure and trustworthy requests are allowed to enter the HTTP stream.
* The system is designed to perform the exploit on both HTTP/1.1 and HTTP/2 protocols, enhancing its versatility and compatibility with different web architectures.
* During the downgrade from HTTP/2 to HTTP/1.1 protocol, the system verifies the headers in the request and only allows the valid headers provided by the administrator.
* By only permitting valid headers, the proposed system significantly enhances the security and reliability of the system, thus minimizing the risk of potential data breaches and unauthorized access.

**3.4 COMPARISON AND ANALYSIS**

**3.4.1 EFFICIENCY**

“Efficiency“ is a critical aspect of any network protocol, including HTTP. To visualize the comparison, we have created a pie chart shown in Figure 3.1. The figure portrays the approximate percentages for each protocol in terms of efficiency are as follows:

* - HTTP/1.0: 25%
* - HTTP/1.1: 50%
* - HTTP/2 : 25%

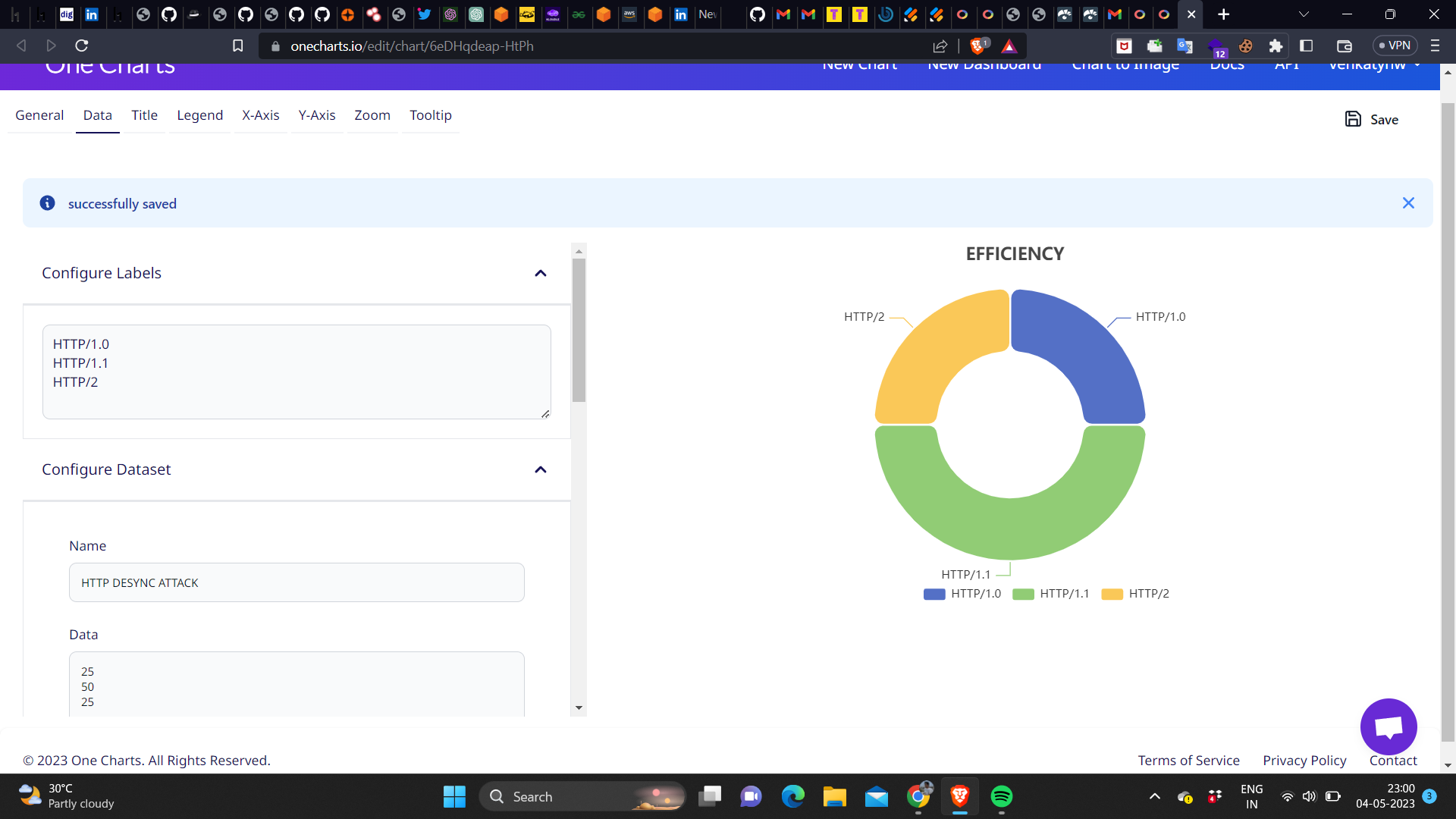
****

Figure 3.1 Efficiency Comparison of HTTP Protocols: HTTP 1.0, 1.1, and 2.0

* It is important to note that these percentages are approximate and depend on the specific use case and implementation. However, they provide a general idea of how the efficiency of these protocols compares to one another.
* HTTP/1.0, the earliest version of HTTP, is the least efficient of the three protocols. It only supports non-persistent connections, meaning that each request/response pair requires a new connection to be established. The proposed approach results in increased overhead, slower data transfer, and higher latency. Additionally, HTTP/1.0 does not support pipelining, which means requests are processed sequentially, further slowing down the processing of requests.
* HTTP/1.1 represents a significant improvement over HTTP/1.0 in terms of efficiency. It supports persistent connections, enabling multiple requests to be sent over a single connection, which reduces the overhead of establishing and tearing down connections. The introduction of pipelining allows multiple requests to be sent simultaneously and processed in parallel, reducing latency and significantly speeding up the processing of requests.
* HTTP/2, the most recent version of HTTP, offers significant efficiency improvements over its predecessors. It introduces several new features, such as multiplexing, which allows multiple requests and responses to be processed simultaneously over a single connection. Additionally, server push enables the server to send resources to the client before they are requested, further reducing latency. HTTP/2 also compresses header data, reducing the amount of data that must be sent with each request and response, thereby improving efficiency.
* While HTTP/1.0 is the least efficient protocol among the three, HTTP/1.1 and HTTP/2 offer significant improvements, with HTTP/2 being the most efficient protocol due to its support for multiplexing, server push, and header compression. The findings of the project can be useful for network administrators and developers to optimize their web applications for better performance and user experience.

**3.4.2 BREAKDOWN OF HTTP PROTOCOL BASED ON USAGE**

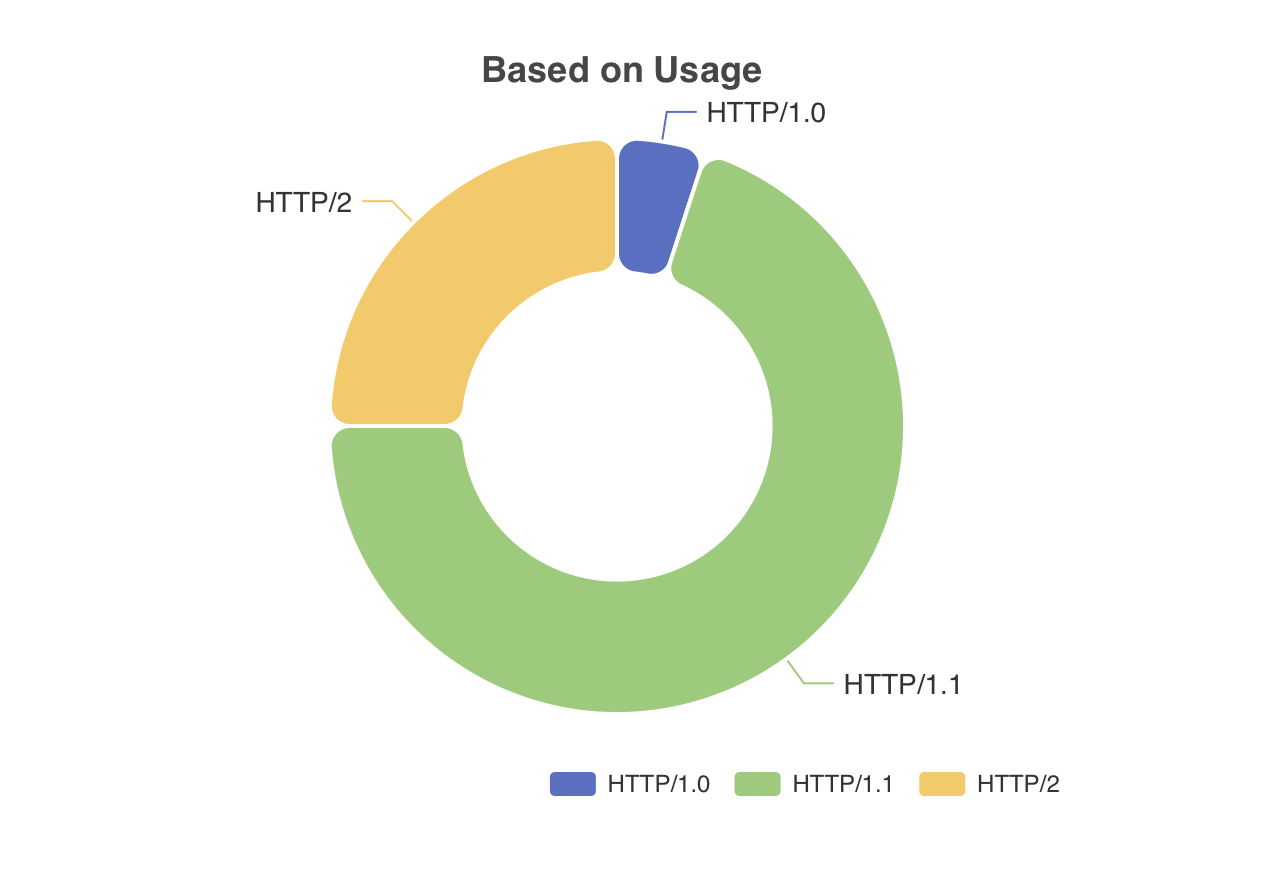
****Figure 3.2 depicts a comparison of the usage percentages of HTTP/1.0, HTTP/1.1, and HTTP/2, highlighting the prevalence of HTTP/1.1 with a usage percentage of about 70%, followed by HTTP/2 with approximately 25% usage. Conversely, HTTP/1.0 accounts for only 5% of website usage, highlighting its inefficiency when compared to newer protocols.

Figure 3.2 Comparison of HTTP Protocols based on usage

* Network protocols play a critical role in the performance of web communication. Among the most widely used protocols are HTTP/1.0, HTTP/1.1, and HTTP/2. The project aims to provide a detailed breakdown of the usage percentages and key features of these protocols that contribute to their efficiency.

Usage Percentages:

* According to recent statistics, HTTP/1.1 is the most widely used protocol in websites today, with a usage percentage of approximately 70%. HTTP/2, the successor to HTTP/1.1, has a usage percentage of around 25%, while HTTP/1.0 only accounts for approximately 5% of website usage.

Key Features:

* HTTP/1.1 was introduced in 1999 and remains the de facto standard for web communication. It features several optimizations over HTTP/1.0, such as support for persistent connections and pipelining. Persistent connections allow multiple requests to be made over a single connection, reducing the overhead associated with establishing and tearing down connections. Pipelining enables multiple requests to be sent simultaneously and processed in parallel, resulting in faster and more efficient communication.
* HTTP/2, introduced in 2015, offers several improvements over its predecessor. Its key features include multiplexing, which allows multiple requests and responses to be processed simultaneously over a single connection. Server push, another key feature, enables the server to push resources to the client before they are requested, further reducing latency. Additionally, HTTP/2 compresses header data, reducing the amount of data that must be sent with each request and response.
* HTTP/1.0 is the oldest protocol still in use today, introduced in 1996. It lacks support for persistent connections or pipelining, which limits its efficiency compared to newer protocols like HTTP/1.1 and HTTP/2.
* HTTP/1.1 remains the most widely used protocol in websites today, but HTTP/2 is rapidly gaining popularity as more websites adopt it. The key features of HTTP/1.1 and HTTP/2, such as persistent connections, pipelining, multiplexing, server push, and header compression, significantly contribute to their efficiency. The findings of the project can be useful for network administrators and developers to optimize their web applications for better performance and user experience.

**3.4.3 COMPARATIVE ANALYSIS OF TOP WAFs FOR PREVENTING HTTP DESYNC ATTACKS**

The Figure 3.3 provides a detailed analysis and comparison of popular Web Application Firewalls (WAFs) suitable for preventing HTTP desync attacks, which exploit vulnerabilities in the way web servers and proxies handle HTTP requests, and can allow attackers to bypass security controls and access sensitive data.

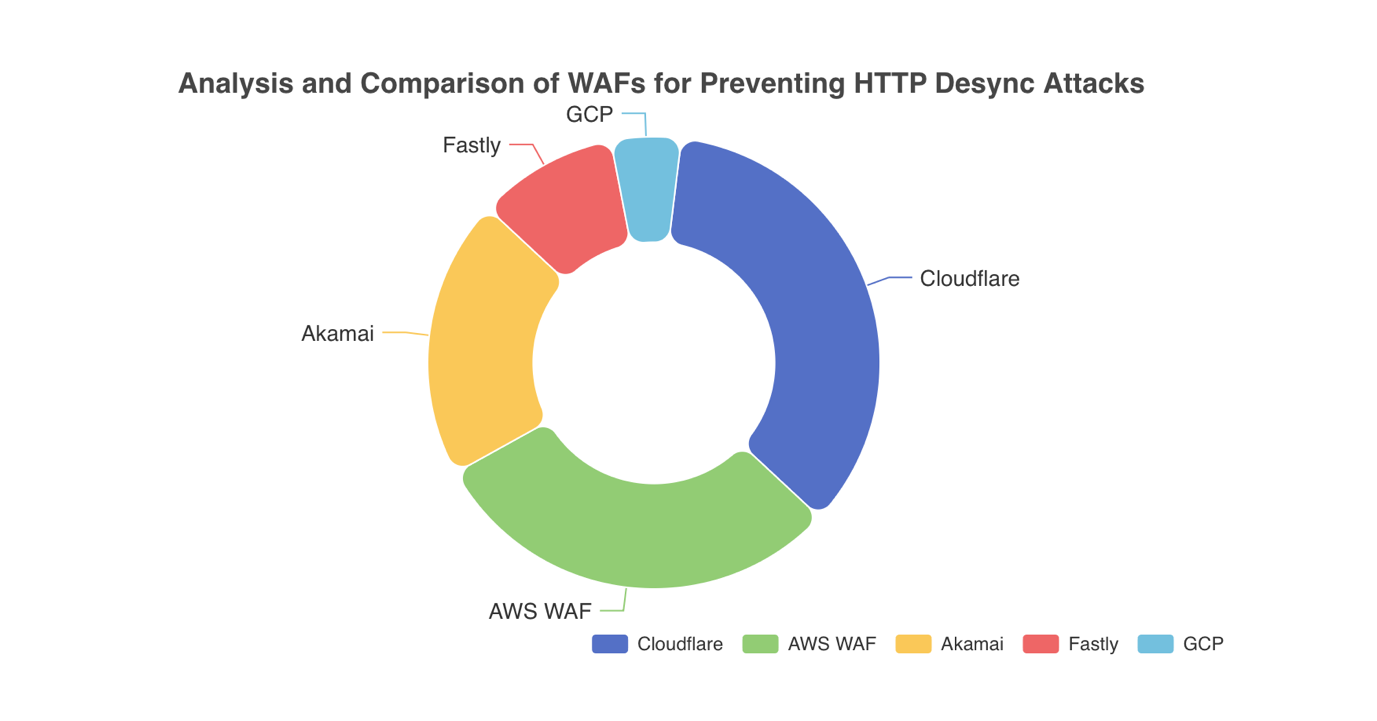
****

Figure 3.3 Top WAFs used in the Industry

* Among the top WAFs suitable for the purpose, Cloudflare leads with a 35% market share. Cloudflare's WAF uses heuristics and algorithms to identify and block desync attacks, and its network architecture is designed to distribute traffic across multiple data centers, making it more resilient to attacks and able to handle large volumes of traffic. AWS WAF follows closely behind, with a 30% market share. AWS WAF is a customizable and scalable solution that can integrate with other AWS services for enhanced protection.
* Akamai, a cloud-based platform that provides a range of security services, including a WAF, holds a 20% market share. Akamai's WAF uses signature-based detection and behavioral analysis to identify and block malicious traffic, and provides advanced reporting and analysis tools to help users identify and respond to attacks in real-time. Fastly, a content delivery network (CDN), holds a 10% market share and offers a WAF service that uses machine learning algorithms to detect and block malicious traffic, and provides real-time updates to ensure that protection is always up-to-date.
* Finally, Google Cloud Platform (GCP) holds a 5% market share and offers a WAF that uses a combination of signature-based detection and machine learning algorithms to identify and block malicious traffic, and provides advanced logging and analysis tools to help users identify and respond to attacks.
* It is important to note that selecting the right WAF for preventing HTTP desync attacks depends on several factors, such as cost, ease of deployment, and specific organizational needs. The report serves as a general guide to the WAFs suitable for preventing HTTP desync attacks as of May 2023 and should not be considered the only resource for making an informed decision.

**CHAPTER 4**

**DESIGN PROCESS**

**4.1 SYSTEM REQUIREMENTS**

**4.1.1 HARDWARE REQUIREMENTS**

Processor : A 64-bit processor with at least two cores

Memory : 8 GB of RAM or more

Storage : At least 50 GB of free hard drive space

Virtualization : WSL requires hardware virtualization support from the CPU

Operating System : Windows 10 or later with the WSL feature enabled

Linux Distribution : A Linux distribution installed in WSL, such as Ubuntu, Debian, or kKali Linux.

**4.1.2 SOFTWARE REQUIREMENTS**

**4.1.2.1 DOCKER**

Docker is a containerization environment used by developers to create and manage lightweight, portable containers for their applications. These containers are self-contained and isolated, making it easy to deploy and scale applications across different environments. Docker offers a range of benefits, including improved development and testing workflows, simplified deployment and management of applications, and increased flexibility and scalability.

One key advantage of Docker is its ability to integrate the frontend and backend components of web applications. In the past, developers had to deploy frontend and backend components separately, which could be complex and time-consuming. However, with Docker, developers can package both the frontend and backend components of an application into a single container, which can be easily deployed and managed.

By using Docker to integrate the frontend and backend components of a web application, developers can streamline their workflows and reduce the complexity of their deployments. Docker provides a consistent, reproducible environment for building and testing applications, ensuring that the application works as intended across different environments. Additionally, Docker offers a range of tools and services for managing and deploying applications, which can simplify the process of scaling and managing web applications.

Another advantage of Docker is its ability to improve the security of web applications. Docker containers are isolated from the host system and from other containers, making it more difficult for attackers to gain unauthorized access to sensitive data or services. Additionally, Docker provides a range of security features, including container image scanning, network security policies, and access control mechanisms, which can help to further enhance the security of web applications.

Moreover, Docker is an open-source platform, which means that developers can benefit from a large community of users and contributors who are constantly improving the platform and creating new tools and services. The community-driven approach enables developers to access a wide range of resources, including documentation, tutorials, and support forums, which can help them to overcome common challenges and improve their skills. Additionally, the open-source nature of Docker allows developers to customize and extend the platform to meet their specific needs, making it a flexible and adaptable solution for a wide range of web application development and deployment scenarios.

In summary, Docker offers developers a powerful set of tools and services for integrating frontend and backend components of web applications. By leveraging Docker containers, developers can streamline their workflows and reduce the complexity of their deployments, leading to more efficient and effective development and deployment of web applications.

**4.1.2.2 BURPSUITE**

Burp Suite Community Edition is a popular web application security testing tool used by developers to identify and mitigate security vulnerabilities in web applications. It offers a range of features, including a proxy server, scanner, and other tools for analyzing web traffic and identifying security issues.

One key advantage of Burp Suite Community Edition is its ability to integrate frontend and backend components of web applications. By intercepting and analyzing web traffic, Burp Suite can identify security vulnerabilities in both the frontend and backend components of web applications, enabling developers to fix issues before they are exploited.

Burp Suite also offers a range of other features, including automated scanning, custom testing scenarios, and support for a variety of platforms and languages. These features enable developers to comprehensively test web applications for security vulnerabilities, providing a more secure and reliable application.

In summary, Burp Suite Community Edition offers developers a powerful set of tools and services for integrating frontend and backend components of web applications. By leveraging its features for web application security testing, developers can identify and mitigate security vulnerabilities in both the frontend and backend components of web applications, leading to more secure and reliable web applications.

The bApps Store is a key feature of Burp Suite, a widely-used web application security testing tool. It provides a range of add-ons, or bApps, that enhance the functionality of the tool and help security testers and developers to more effectively identify and address vulnerabilities in web applications.

The bApps Store includes a diverse collection of bApps, created both by PortSwigger, the developers of Burp Suite, and third-party developers. These bApps expand the capabilities of Burp Suite, offering new tools for scanning, reporting, and automating security tests.

Some of the most popular bApps available in the bApps Store are ActiveScan++, a sophisticated scanner that identifies various web application vulnerabilities; AuthMatrix, which simplifies the testing of authentication and authorization mechanisms; Logger++, a powerful logging tool for capturing and analyzing HTTP traffic; and Retire.js, which detects outdated and potentially exploitable JavaScript libraries in web applications.

Users can easily download and install bApps from the bApps Store within Burp Suite, and developers can also create and share their own bApps through the platform, contributing to the ongoing growth and development of the tool.

**4.1.2.3 WEB BROWSER**

Web browsers such as Mozilla Firefox and Google Chrome are essential tools for accessing and is essential in navigating the internet. They offer a range of features, including tabbed browsing, bookmarking, and extensions, that allow users to customize their browsing experience.

One key advantage of these web browsers is their support for web standards, ensuring that web pages are rendered accurately and consistently across different devices and platforms. They also offer security features, such as anti-phishing and anti-malware protection, to protect users from online threats.

Mozilla Firefox and Google Chrome also offer developer tools that allow web developers to debug and optimize their web applications. These tools provide insights into page performance, network activity, and rendering, allowing developers to identify and address issues that affect the user experience.

Web browsers can also serve as a testing environment for sending HTTP requests. Developers and security testers can use tools like the browser's developer console or extensions like Postman to send HTTP requests and view the responses in real-time. It allows for easier testing and debugging of web applications and APIs.

Additionally, web browsers support various HTTP methods, such as GET, POST, PUT, DELETE, and more, which can be utilized in testing and development. Developers can send HTTP requests to their own web applications or third-party services to verify that the expected data is being returned.

Furthermore, the ability to inspect and modify HTTP headers and cookies in the browser can be useful in testing the security of web applications. For instance, testers can manipulate cookies to test for vulnerabilities like session hijacking.

Overall, web browsers provide a convenient and accessible way to test web applications and APIs by allowing users to send HTTP requests and view responses in real-time, and by providing tools to inspect and modify HTTP headers and cookies.

**4.1.2.4 TURBO INTRUDER**

Turbo Intruder is a powerful and flexible HTTP/S testing tool used for security testing and penetration testing. It is built in Python and has a number of features that make it a popular choice among security researchers and pentesters.

One of the key features of Turbo Intruder is its speed. It is designed to be extremely fast and efficient, allowing you to test a large number of targets quickly and accurately. It is achieved by using multithreading and other techniques to maximize the throughput of the tool.

Another feature of Turbo Intruder is its flexibility. It allows you to easily customize the requests and responses, making it easy to test specific scenarios and attack vectors. It includes support for regular expressions, macros, and other advanced features that are often required for complex testing scenarios.

Turbo Intruder also has a number of built-in tools and plugins that make it easy to use with other testing tools and frameworks. For example, it has built-in support for Burp Suite, a popular web application testing tool, and can be easily integrated with other tools and frameworks.

In addition to its speed and flexibility, Turbo Intruder also offers a user-friendly interface that makes it easy to use for both beginners and experienced security professionals. The tool provides a simple and intuitive interface for configuring and running tests, making it easy to customize requests and responses and quickly identify vulnerabilities. Turbo Intruder also generates detailed reports that provide a comprehensive overview of the testing results, allowing you to easily identify and prioritize vulnerabilities for remediation.

Another advantage of Turbo Intruder is its support for automation and scripting. The tool can be easily integrated with other automation frameworks and scripting languages, allowing you to automate repetitive testing tasks and scale the testing efforts. It can save time and effort while ensuring that testing is thorough and comprehensive. Additionally, the tool can be run from the command line, making it easy to integrate with other tools and processes.

In summary, Turbo Intruder is a powerful and versatile tool that offers a range of features and capabilities for security testing and penetration testing. Its speed, flexibility, user-friendly interface, and support for automation and scripting make it an ideal choice for security professionals and pentesters who need to quickly and effectively test web applications for vulnerabilities. By leveraging the capabilities of Turbo Intruder, security professionals can ensure the security and reliability of their web applications, protecting both their users and their own interests.

**4.2 MODULE EXPLANATION**

**4.2.1 MODULE 1 - DEPLOYMENT**

* The Deployment module's responsibility is to implement HTTP desync attack techniques in a web application or server environment.
* A testing environment has been developed to facilitate it, which includes various web servers, clients, and pre-configured environments for executing HTTP desync attacks.
* The testing environment is based on Docker containers and can be easily deployed using Docker Compose.
* The containers are configured to simulate real-world web application scenarios and can be integrated into an existing web application or server environment.
* To integrate the testing environment, the user needs to install Docker and Docker Compose on the host system, and then execute the Docker Compose command to launch the containers.
* After launching the containers, the web application running inside them can be accessed using a web browser or an HTTP client tool.
* Doing it allows the user to execute various HTTP desync attacks against the web application and test its resilience to these attacks. The Deployment module is essential for testing and evaluating the effectiveness of HTTP desync attack countermeasures.
* By using the testing environment and integrating it into a testing environment, users can effectively simulate real-world attack scenarios and evaluate the security of web applications and servers.

**4.2.2 MODULE 2 - ANALYSIS**

• The Analysis module is responsible for developing a Python tool that can detect HTTP desync attacks. The tool uses various techniques to detect these attacks and does not require a proxy to check incoming requests.

• The Python tool can be installed and executed on any system with Python installed. To use it, the user needs to provide the URL of the web application and test it. The tool then sends requests to the web application and analyzes the responses to detect any signs of HTTP desync attacks.

• The tool is designed to be user-friendly and provides detailed reports on the results of each test. It allows developers to quickly identify any vulnerabilities that need to be addressed and take appropriate action.

• Overall, the analysis module is an essential component of the HTTP desync attack testing framework. By using the detection tool, users can effectively test the resilience of web applications to these attacks and gain valuable insights for improving their security.

**4.2.3 MODULE 3 - DEMONSTRATION**

• The Demonstration module is responsible for demonstrating the impact of HTTP desync attacks on a web application. For the Demonstration module, Burp Suite Turbo Intruder was used to exploit the web application and test its resilience to these attacks. A Python script was created that can be used with Turbo Intruder to automate the process of sending multiple requests and payloads to the web application.

• To use the Python script with Turbo Intruder, the user first needs to configure the request and payload settings in Turbo Intruder. The script is then provided to Turbo Intruder, which executes the script to send requests and payloads to the web application.

• The Python script developed by the Demonstration module uses various techniques to generate payloads and sends multiple requests to the web application to test its resilience to HTTP desync attacks. The script provides detailed reports on the results of each test, which can be used to identify vulnerabilities and improve the security of the web application.

• Overall, the Demonstration module is a critical component of the HTTP desync attack testing framework. By demonstrating the impact of these attacks on a web application, awareness can be raised on the importance of implementing effective countermeasures to protect against these types of attacks.

**4.2.4 MODULE 4 - PROTECTION**

• It is important to keep the web server and proxy software updated to the latest version to address known vulnerabilities that may be exploited by attackers.

• Specialized security tools such as web traffic monitoring and analysis tools can be used to detect and mitigate HTTP desync attacks.

• Strict HTTP request parsing rules should be implemented to prevent HTTP desync attacks.

• WAF rules are a set of guidelines or instructions that are used by a security tool called a Web Application Firewall. The WAF rules tell the firewall what kind of traffic should be allowed through to a website or web application, and what kind of traffic should be blocked.

• For example, if someone tries to access a website with a known vulnerability, the WAF rules can be set up to block that traffic, preventing potential hackers from exploiting the vulnerability and gaining access to sensitive data.

• A web application firewall (WAF) should be deployed to protect web applications from various application layer attacks including HTTP desync attacks.

• A WAF can help detect and prevent HTTP desync attacks by analyzing HTTP requests and responses for anomalies and blocking suspicious traffic.

• WAF rules should be regularly reviewed and updated to ensure effective protection against evolving attack methods.

**4.2.5 MODULE 5 - UPDATES**

• A Web Application Firewall is a security tool designed to protect web applications from malicious attacks by closely monitoring and filtering incoming traffic.

• New WAF rules can be added to the firewall as new threats are identified, ensuring that web applications are secure and protected against attacks.

• Enhancements to the security of web applications have been made through the use of fundamental techniques and the latest WAF rules.

• WAF rules act as a set of policies that dictate which traffic is allowed or blocked by the firewall. Fresh WAF rules are developed to counter specific types of traffic or behavior that may pose a threat to the system.

• A GitHub repository serves as a centralized location for new updates and rulesets, allowing users to download them and stay up-to-date with the latest security measures.

**4.3 SYSTEM ARCHITECTURE**

The H2.CL attack is a serious threat to the security of web applications, as it exploits a vulnerability in the handling of HTTP/2 requests. In the H2.CL attack, an attacker smuggles a malicious request into a victim's legitimate request, bypassing security controls and gaining access to sensitive information. Understanding the system architecture involved in the attack is crucial to developing effective security measures. Figure 4.1 illustrates the underlying system architecture of how the request is sent and is responded respectively.

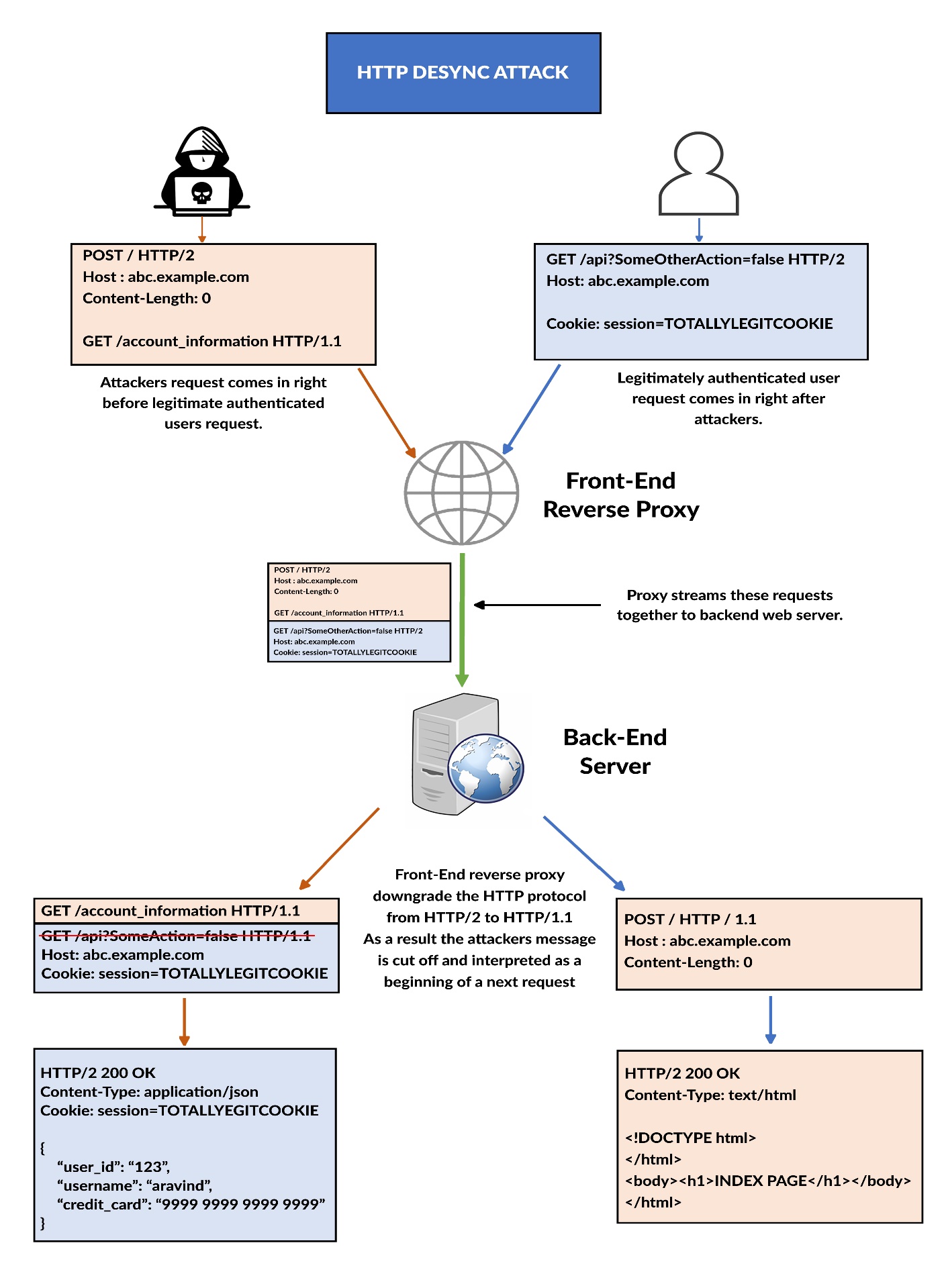


Figure 4.1 System Architecture

The system architecture described in Figure 4.1 involves two users, an attacker and a victim, both sending requests to a frontend server. The attacker is not authorized and not logged in, while the victim has logged in and has a cookie for his session. Both users send their requests to the frontend server, with the attacker using the POST method and sending a request body with a Content-Length header set to 0. The attacker has also placed a GET request within the body of the POST request, asking for the victim's account information. Meanwhile, the victim is requesting a website using the GET request.

When both requests are sent to the frontend server, the attacker's request is prioritized and processed before the victim's request. Both requests utilize the HTTP/2 protocol initially, but as the frontend server forwards them to the backend, a downgrade to the HTTP/1.1 protocol occurs. The frontend server's role is to add a Content-Length header if it is absent in the request before passing it to the backend. However, in the attacker's request, a Content-Length header already exists, preventing the frontend server from adding any additional headers. Consequently, the frontend server downgrades the protocol to HTTP/1.1 and forwards the request.

At the backend, the downgraded message is received and evaluated. Despite a Content-Length of 0, there is content present in the body section of the message (GET /account\_information HTTP/1.1). Mistakenly considering it as a separate request, the backend treats the attacker's message as the start of a new request. Consequently, the backend observes two GET requests but is programmed to only accept the first valid header, neglecting the second request (as depicted in Figure 4.1). Subsequently, the attacker receives the response to the manipulated request from the backend, which combines a GET request with the victim's Cookie. The attacker gains unauthorized access to sensitive information of the victim's Account.

However, it is important to note that the response for the attacker's initial request, which was for a webpage (abc.example.com), does not match the expected result. The desynchronization between the client and the server is the reason why it is referred to as an HTTP Desync attack. It highlights a deviation from the intended behavior of a well-functioning system architecture, where legitimate requests are appropriately processed, and authorized users can access the resources they require.

In summary, the system architecture described in Figure 4.1 highlights the vulnerability in the HTTP/2 protocol where an attacker can exploit a server's downgrade to HTTP/1.1 to inject malicious requests and retrieve sensitive information. The vulnerability can be addressed through implementing security measures such as encrypting all traffic, enforcing strict header validation, and blocking requests with invalid headers.

Another strategy to improve security is to adopt a layered security model that involves implementing various security controls at different levels of the system architecture. For instance, network-level security controls like firewalls and intrusion detection systems can be used to prevent unauthorized access to the system, while application-level security controls such as input validation and access controls can be utilized to prevent malicious requests from being executed and sensitive data from being accessed.

Periodic security audits and vulnerability assessments are essential to detect and mitigate security weaknesses in the system architecture. These assessments involve analyzing data and graphs to identify patterns of attacks and develop effective countermeasures. By doing so, security teams can stay ahead of potential threats and prevent security breaches before they occur. It is crucial to incorporate these assessments as part of a proactive approach to security rather than waiting for an attack to occur. By being proactive, organizations can reduce the likelihood and impact of security breaches, ensuring the safety of their systems and data.

**CHAPTER 5**

**IMPLEMENTATION**

For the implementation of the HTTP desync attack testing framework, various types of web servers were used as both frontend and backend components. The servers were configured to simulate real-world web application scenarios, and techniques such as modifying server configurations, adding custom headers, and changing response codes were employed to configure them. Wireshark was used to capture and analyze network traffic to ensure the servers responded appropriately to the HTTP desync attack techniques.

To set up a testing environment that was consistent across different systems, Docker containers were used, and Docker Compose was employed to manage the containers. A Python detection tool was used to test the web servers for HTTP desync attacks, and Burp Suite Turbo Intruder, along with a custom Python script, was used to test the servers for vulnerabilities and exploit any weaknesses found.

In addition to the techniques and tools used to test the web servers, the implementation also included comprehensive documentation and reporting of the testing results. The documentation served as a valuable resource for developers and security professionals, enabling them to identify and fix vulnerabilities and strengthen the security of web applications. The reporting also provided insights into the effectiveness of the HTTP desync attack testing framework, helping to improve its capabilities and identify areas for future development.

Moreover, the implementation demonstrated the importance of proactive testing and security measures for web applications. As the number and complexity of web-based services continue to increase, the need for robust and effective security measures becomes more critical. By using tools and techniques like those employed in the HTTP desync attack testing framework, organizations can ensure the security and reliability of their web applications, protecting both their users and their own interests. Therefore, the implementation serves as a valuable example and guide for developing effective testing frameworks and enhancing web application security.

Overall, the implementation successfully simulated real-world web application scenarios and tested the effectiveness of the HTTP desync attack testing framework. By using Wireshark and other testing tools, the framework was ensured to be robust and effective in identifying and mitigating HTTP desync attacks while maintaining the security of the web applications.

**CHAPTER 6**

**CONCLUSION AND FUTURE ENHANCEMENT**

**6.1 CONCLUSION**

It proposes an effective methodology for testing web server vulnerability to HTTP desync attacks in both HTTP/1.x and HTTP/2 protocols. The use of specialized security tools such as Turbo Intruder and Python scripts enables the detection and mitigation of HTTP desync attacks. Findings suggest that implementing strict HTTP request parsing rules and keeping web server and proxy software up-to-date can enhance the prevention of HTTP desync attacks.

In summary, the comprehensive approach provides a viable solution for detecting and preventing HTTP desync attacks in web applications, which could help security professionals to improve their testing practices and safeguard web applications from an increasingly common attack vector.

**6.2 FUTURE ENHANCEMENT**

For future works, it is recommended to explore the potential of incorporating machine learning and artificial intelligence algorithms in detecting and preventing HTTP desync attacks. Additionally, the investigation of alternative WAF technologies and a comparative analysis of their effectiveness in preventing HTTP desync attacks could be valuable areas for further research. Furthermore, analyzing the impact of HTTP desync attacks on various applications, protocols, and platforms would provide insights into their broader implications. The project aims to establish a foundation for further research in web application security and contribute to the development of more effective measures against HTTP desync attacks.

**REFERENCES**

1. Min-Yi Chiu, Ying-Feng Chen, and Hung-Min Sun., “Attacking Websites: Detecting and Preventing HTTP Request Smuggling Attacks”, Taiwan, October 2022

2. C. Linhart, “HTTP-Request-Smuggling”, Portswigger, Chelford Road, Knutsford, 2005.

3. B. Jabiyev, S. Sprecher, K. Onarlioglu, and E. Kirda, “T-reqs: HTTP request smuggling with differential fuzzing,” in proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS’21), November 2021.

4. J. Kettle, “HTTP Desync Attacks: Smashing into the Cell Next Door”, Portswigger, Black Hat USA, 2019.

5. R. Fielding and J. Reschke, “Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content”, https://www.rfc-editor.org/rfc/rfc7231.txt, 2014.

6. A. Klein, “HTTP Request Smuggling In 2020 – New Variants, New Defenses and New Challenges”, Portswigger, Black Hat USA, 2020.

7. M. Grenfeldt and A. Olofsson, “Empirical Study of HTTP Request Smuggling in Open-Source Servers and Proxies”, June 2021.

8. Régis Leroy., “Hiding Wookies in HTTP”, DEF CON 24. Aug.2016

9. Christoph Kern. “Securing the tangled web”. In: Communications of the ACM 57.9 (2014), pp. 38–47.

10. Gareth Tyson et al. “Exploring http header manipulation in-the-wild”.In: Proceedings of the 26th International Conference on World Wide Web. 2017, pp. 451–458.

11. M. A. Wazzan and M. H. Awadh, “Towards improving web attack detection: highlighting the significant factors,” in Proceedings of the 2015 5th International Conference on IT Convergence and Security (ICITCS), pp. 1–5, IEEE, Kuala Lumpur, Malaysia, August 2015.

12. R. Leroy, “Checking HTTP Smuggling issues in 2015 - Part1,” 2015, http://regilero.github.io/security/english/2015/10/04/http\_smuggling\_in\_2015\_part\_one/.

13. M. Doyhenard, “Response Smuggling: Exploiting HTTP/1.1 Connections” HTTP Response Smuggling - DEFCON paper.

14. R. Fielding and J. Reschke, “Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing”, https://www.rfc-editor.org/rfc/rfc7231.txt, 2014.

15. Jon Postel., “Transmission Control Protocol”. http://www.rfc-editor.org/rfc/rfc793.txt.,1981.

16. S. A. Mirheidari, et.al. “Web Cache Deception Escalates” at 31st USENIX Security Symposium. Boston, MA, August 2022.

17. H. Nguyen et al. “Your Cache Has Fallen: Cache-Poisoned Denial-of-Service Attack” at 26th ACM Conference on Computer and Communications Security (CCS), London, UK, 2019

**CHAPTER 8**

**APPENDIX A1**

**SOURCE CODE**

**docker-compose.yml**

version: '3'

services:

#Armeria Proxy

armeria:

build:

context: .

dockerfile: ./Dockerfile-armeria

ports:

- "8443:8443"

webmain:

image: php:8.0.13-apache

volumes:

- ./webmain/:/var/www/html/

ports:

- "8001:80"

webstatic:

image: php:8.0.13-apache

volumes:

- ./webstatic/:/var/www/html/

ports:

- "8002:80"

**DOCKERFILE-ARMERIA**

FROM frolvlad/alpine-java

VOLUME /tmp

COPY armeria/vulnerable-armeria-all.jar proxy.jar

RUN sh -c 'touch /proxy.jar'

ENTRYPOINT [“java","-Djava.security.egd=file:/dev/./urandom","-jar","/proxy.jar"]

**WEBSTATIC - DOCUMENTS**

This document should not be exposed.

flag{documentsAccessGranted}

**Desyncron Tool :**

import argparse

import re

import time

import sys

import os

import random

import string

import importlib

import hashlib

from copy import deepcopy

from time import sleep

from datetime import datetime

from lib.Payload import Payload, Chunked, EndChunk

from lib.EasySSL import EasySSL

from lib.colorama import Fore, Style

from urllib.parse import urlparse

class Desyncr():

def \_\_init\_\_(self, configfile, smhost, smport=443, url="", method="POST", endpoint="/", SSLFlag=False, logh=None,

smargs=None):

self.\_configfile = configfile

self.\_host = smhost

self.\_port = smport

self.\_method = method

self.\_endpoint = endpoint

self.\_vhost = smargs.vhost

self.\_url = url

self.\_timeout = float(smargs.timeout)

self.ssl\_flag = SSLFlag

self.\_logh = logh

self.\_quiet = smargs.quiet

self.\_exit\_early = smargs.exit\_early

self.\_attempts = 0

self.\_cookies = []

def \_test(self, payload\_obj):

try:

web = EasySSL(self.ssl\_flag)

web.connect(self.\_host, self.\_port, self.\_timeout)

web.send(str(payload\_obj).encode())

# print(payload\_obj)

start\_time = datetime.now()

res = web.recv\_nb(self.\_timeout)

end\_time = datetime.now()

web.close()

if res is None:

delta\_time = end\_time - start\_time

if delta\_time.seconds < (self.\_timeout - 1):

return (2, res, payload\_obj) # Return code 2 if disconnected before timeout

return (1, res, payload\_obj) # Return code 1 if connection timedout

# Filter out problematic characters

res\_filtered = ""

for single in res:

if single > 0x7F:

res\_filtered += '\x30'

else:

res\_filtered += chr(single)

res = res\_filtered

# if '504' in res:

# print("\n\n"+str(str(payload\_obj)))

# print("\n\n"+res)

return (0, res, payload\_obj) # Return code 0 if normal response returned

except Exception as exception\_data:

# print(exception\_data)

return (-1, None, payload\_obj) # Return code -1 if some except occured

def \_get\_cookies(self):

RN = "\r\n"

try:

cookies = []

web = EasySSL(self.ssl\_flag)

web.connect(self.\_host, self.\_port, 2.0)

p = Payload()

p.host = self.\_host

p.method = "GET"

p.endpoint = self.\_endpoint

p.header = "\_\_METHOD\_\_ \_\_ENDPOINT\_\_?cb=\_\_RANDOM\_\_ HTTP/1.1" + RN

p.header += "Host: \_\_HOST\_\_" + RN

p.header += "User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/78.0.3904.87 Safari/537.36" + RN

p.header += "Content-type: application/x-www-form-urlencoded; charset=UTF-8" + RN

p.header += "Content-Length: 0" + RN

p.body = ""

# print (str(p))

web.send(str(p).encode())

sleep(0.5)

res = web.recv\_nb(2.0)

web.close()

if (res is not None):

res = res.decode().split("\r\n")

for elem in res:

if len(elem) > 11:

if elem[0:11].lower().replace(" ", "") == "set-cookie:":

cookie = elem.lower().replace("set-cookie:", "")

cookie = cookie.split(";")[0] + ';'

cookies += [cookie]

info = ((Fore.CYAN + str(len(cookies)) + Fore.MAGENTA), self.\_logh)

print\_info("Cookies : %s (Appending to the attack)" % (info[0]))

self.\_cookies += cookies

return True

except Exception as exception\_data:

error = ((Fore.CYAN + "Unable to connect to host" + Fore.MAGENTA), self.\_logh)

print\_info("Error : %s" % (error[0]))

return False

def run(self):

RN = "\r\n"

mutations = {}

if not self.\_get\_cookies():

return

if (self.\_configfile[1] != '/'):

self.\_configfile = os.path.dirname(os.path.realpath(\_\_file\_\_)) + "/configs/" + self.\_configfile

try:

f = open(self.\_configfile)

except:

error = ((Fore.CYAN + "Cannot find config file" + Fore.MAGENTA), self.\_logh)

print\_info("Error : %s" % (error[0]))

exit(1)

script = f.read()

f.close()

exec(script)

for mutation\_name in mutations.keys():

if self.\_create\_exec\_test(mutation\_name, mutations[mutation\_name]) and self.\_exit\_early:

break

if self.\_quiet:

sys.stdout.write("\r" + " " \* 100 + "\r")

# ptype == 0 (Attack payload, timeout could mean potential TECL desync)

# ptype == 1 (Edgecase payload, expected to work)

def \_check\_tecl(self, payload, ptype=0):

te\_payload = deepcopy(payload)

if (self.\_vhost == ""):

te\_payload.host = self.\_host

else:

te\_payload.host = self.\_vhost

te\_payload.method = self.\_method

te\_payload.endpoint = self.\_endpoint

if len(self.\_cookies) > 0:

te\_payload.header += "Cookie: " + ''.join(self.\_cookies) + "\r\n"

if not ptype:

te\_payload.cl = 6 # timeout val == 6, good value == 5

else:

te\_payload.cl = 5 # timeout val == 6, good value == 5

te\_payload.body = EndChunk + "X"

# print (te\_payload)

return self.\_test(te\_payload)

# ptype == 0 (timeout payload, timeout could mean potential CLTE desync)

# ptype == 1 (Edgecase payload, expected to work)

def \_check\_clte(self, payload, ptype=0):

te\_payload = deepcopy(payload)

if (self.\_vhost == ""):

te\_payload.host = self.\_host

else:

te\_payload.host = self.\_vhost

te\_payload.method = self.\_method

te\_payload.endpoint = self.\_endpoint

if len(self.\_cookies) > 0:

te\_payload.header += "Cookie: " + ''.join(self.\_cookies) + "\r\n"

if not ptype:

te\_payload.cl = 4 # timeout val == 4, good value == 11

else:

te\_payload.cl = 11 # timeout val == 4, good value == 11

te\_payload.body = Chunked("Z") + EndChunk

# print (te\_payload)

return self.\_test(te\_payload)

def \_create\_exec\_test(self, name, te\_payload):

def pretty\_print(name, dismsg):

spacing = 13

sys.stdout.write("\r" + " " \* 100 + "\r")

msg = Style.BRIGHT + Fore.MAGENTA + "[%s]%s: %s" % \

(Fore.CYAN + name + Fore.MAGENTA, " " \* (spacing - len(name)), dismsg)

sys.stdout.write(CF(msg + Style.RESET\_ALL))

sys.stdout.flush()

if dismsg[-1] == "\n":

ansi\_escape = re.compile(r'\x1B[@-\_][0-?]\*[ -/]\*[@-~]')

plaintext = ansi\_escape.sub('', msg)

if self.\_logh is not None:

self.\_logh.write(plaintext)

self.\_logh.flush()

def write\_payload(smhost, payload, ptype):

furl = smhost.replace('.', '\_')

if (self.ssl\_flag):

furl = "https\_" + furl

else:

furl = "http\_" + furl

if os.path.islink(sys.argv[0]):

\_me = os.readlink(sys.argv[0])

else:

\_me = sys.argv[0]

fname = os.path.realpath(os.path.dirname(\_me)) + "/payloads/%s\_%s\_%s.txt" % (furl, ptype, name)

pretty\_print("CRITICAL", "%s Payload: %s URL: %s\n" % \

(Fore.MAGENTA + ptype, Fore.CYAN + fname + Fore.MAGENTA, Fore.CYAN + self.\_url))

with open(fname, 'wb') as file:

file.write(bytes(str(payload), 'utf-8'))

# First lets test TECL

pretty\_print(name, "Checking TECL...")

start\_time = time.time()

tecl\_res = self.\_check\_tecl(te\_payload, 0)

tecl\_time = time.time() - start\_time

# Next lets test CLTE

pretty\_print(name, "Checking CLTE...")

start\_time = time.time()

clte\_res = self.\_check\_clte(te\_payload, 0)

clte\_time = time.time() - start\_time

if (clte\_res[0] == 1):

# Potential CLTE found

# Lets check the edge case to be sure

clte\_res2 = self.\_check\_clte(te\_payload, 1)

if clte\_res2[0] == 0:

self.\_attempts += 1

if (self.\_attempts < 3):

return self.\_create\_exec\_test(name, te\_payload)

else:

dismsg = Fore.RED + "Potential CLTE Issue Found" + Fore.MAGENTA + " - " + Fore.CYAN + self.\_method + Fore.MAGENTA + " @ " + Fore.CYAN + \

["http://", "https://", ][

self.ssl\_flag] + self.\_host + self.\_endpoint + Fore.MAGENTA + " - " + Fore.CYAN + \

self.\_configfile.split('/')[-1] + "\n"

pretty\_print(name, dismsg)

# Write payload out to file

write\_payload(self.\_host, clte\_res[2], "CLTE")

self.\_attempts = 0

return True

else:

# No edge behavior found

dismsg = Fore.YELLOW + "CLTE TIMEOUT ON BOTH LENGTH 4 AND 11" + ["\n", ""][self.\_quiet]

pretty\_print(name, dismsg)

elif (tecl\_res[0] == 1):

# Potential TECL found

# Lets check the edge case to be sure

tecl\_res2 = self.\_check\_tecl(te\_payload, 1)

if tecl\_res2[0] == 0:

self.\_attempts += 1

if (self.\_attempts < 3):

return self.\_create\_exec\_test(name, te\_payload)

else:

# print (str(tecl\_res2[2]))

# print (tecl\_res2[1])

dismsg = Fore.RED + "Potential TECL Issue Found" + Fore.MAGENTA + " - " + Fore.CYAN + self.\_method + Fore.MAGENTA + " @ " + Fore.CYAN + \

["http://", "https://", ][

self.ssl\_flag] + self.\_host + self.\_endpoint + Fore.MAGENTA + " - " + Fore.CYAN + \

self.\_configfile.split('/')[-1] + "\n"

pretty\_print(name, dismsg)

# Write payload out to file

write\_payload(self.\_host, tecl\_res[2], "TECL")

self.\_attempts = 0

return True

else:

# No edge behavior found

dismsg = Fore.YELLOW + "TECL TIMEOUT ON BOTH LENGTH 6 AND 5" + ["\n", ""][self.\_quiet]

pretty\_print(name, dismsg)

# elif ((tecl\_res[0] == 1) and (clte\_res[0] == 1)):

# # Both types of payloads not supported

# dismsg = Fore.YELLOW + "NOT SUPPORTED" + ["\n", ""][self.\_quiet]

# pretty\_print(name, dismsg)

elif ((tecl\_res[0] == -1) or (clte\_res[0] == -1)):

# ERROR

dismsg = Fore.YELLOW + "SOCKET ERROR" + ["\n", ""][self.\_quiet]

pretty\_print(name, dismsg)

elif ((tecl\_res[0] == 0) and (clte\_res[0] == 0)):

# No Desync Found

tecl\_msg = (Fore.MAGENTA + " (TECL: " + Fore.CYAN + "%.2f" + Fore.MAGENTA + " - " + \

Fore.CYAN + "%s" + Fore.MAGENTA + ")") % (tecl\_time, tecl\_res[1][9:9 + 3])

clte\_msg = (Fore.MAGENTA + " (CLTE: " + Fore.CYAN + "%.2f" + Fore.MAGENTA + " - " + \

Fore.CYAN + "%s" + Fore.MAGENTA + ")") % (clte\_time, clte\_res[1][9:9 + 3])

dismsg = Fore.GREEN + "OK" + tecl\_msg + clte\_msg + ["\n", ""][self.\_quiet]

pretty\_print(name, dismsg)

elif ((tecl\_res[0] == 2) or (clte\_res[0] == 2)):

# Disconnected

dismsg = Fore.YELLOW + "DISCONNECTED" + ["\n", ""][self.\_quiet]

pretty\_print(name, dismsg)

self.\_attempts = 0

return False

def process\_uri(uri):

u = urlparse(uri)

if u.scheme == "https":

ssl\_flag = True

std\_port = 443

elif u.scheme == "http":

ssl\_flag = False

std\_port = 80

else:

print\_info("Error malformed URL not supported: %s" % (Fore.CYAN + uri))

exit(1)

if u.port:

return (u.hostname, u.port, u.path, ssl\_flag)

else:

return (u.hostname, std\_port, u.path, ssl\_flag)

def CF(text):

global NOCOLOR

if NOCOLOR:

ansi\_escape = re.compile(r'\x1B[@-\_][0-?]\*[ -/]\*[@-~]')

text = ansi\_escape.sub('', text)

return text

def banner(sm\_version):

print(CF(Fore.CYAN))

print(CF(r" \_\_\_\_ "))

print(CF(r"| \_ \ \_\_\_ \_\_\_ \_ \_ \_ \_\_ \_\_\_ \_ \_\_ \_\_\_ \_ \_\_ "))

print(CF(r"| | | |/ \_ \/ \_\_| | | | '\_ \ / \_\_| '\_\_/ \_ \| '\_ \ "))

print(CF(r"| |\_| | \_\_/\\_\_ \ |\_| | | | | (\_\_| | | (\_) | | | |"))

print(CF(r"|\_\_\_\_/ \\_\_\_||\_\_\_/\\_\_, |\_| |\_|\\_\_\_|\_| \\_\_\_/|\_| |\_|"))

print(CF(r" |\_\_\_/ "))

print(CF(r" %s" % (sm\_version)))

print(CF(Style.RESET\_ALL))

def print\_info(msg, file\_handle=None):

ansi\_escape = re.compile(r'\x1B[@-\_][0-?]\*[ -/]\*[@-~]')

msg = Style.BRIGHT + Fore.MAGENTA + "[%s] %s" % (Fore.CYAN + '+' + Fore.MAGENTA, msg) + Style.RESET\_ALL

plaintext = ansi\_escape.sub('', msg)

print(CF(msg))

if file\_handle is not None:

file\_handle.write(plaintext + "\n")

if \_\_name\_\_ == "\_\_main\_\_":

global NOCOLOR

if sys.version\_info < (3, 0):

print("Error: Smuggler requires Python 3.x")

sys.exit(1)

Parser = argparse.ArgumentParser()

Parser.add\_argument('-u', '--url', help="Target URL with Endpoint")

Parser.add\_argument('-v', '--vhost', default="", help="Specify a virtual host")

Parser.add\_argument('-x', '--exit\_early', action='store\_true', help="Exit scan on first finding")

Parser.add\_argument('-m', '--method', default="POST", help="HTTP method to use (e.g GET, POST) Default: POST")

Parser.add\_argument('-l', '--log', help="Specify a log file")

Parser.add\_argument('-q', '--quiet', action='store\_true', help="Quiet mode will only log issues found")

Parser.add\_argument('-t', '--timeout', default=5.0, help="Socket timeout value Default: 5")

Parser.add\_argument('--no-color', action='store\_true', help="Suppress color codes")

Parser.add\_argument('-c', '--configfile', default="default.py",

help="Filepath to the configuration file of payloads")

Args = Parser.parse\_args() # returns data from the options specified (echo)

NOCOLOR = Args.no\_color

if os.name == 'nt':

NOCOLOR = True

Version = "v1.1"

banner(Version)

if sys.version\_info < (3, 0):

print\_info("Error: Smuggler requires Python 3.x")

sys.exit(1)

# If the URL argument is not specified then check stdin

if Args.url is None:

if sys.stdin.isatty():

print\_info("Error: no direct URL or piped URL specified\n")

Parser.print\_help()

exit(1)

Servers = sys.stdin.read().split("\n")

else:

Servers = [Args.url + " " + Args.method]

FileHandle = None

if Args.log is not None:

try:

FileHandle = open(Args.log, "w")

except:

print\_info("Error: Issue with log file destination")

print(Parser.print\_help())

sys.exit(1)

for server in Servers:

# If the next on the list is blank, continue

if server == "":

continue

# Tokenize

server = server.split(" ")

# This is for the stdin case, if no method was specified default to GET

if len(server) == 1:

server += [Args.method]

# If a protocol is not specified then default to https

if server[0].lower().strip()[0:4] != "http":

server[0] = "https://" + server[0]

host, port, endpoint, SSLFlagval = process\_uri(server[0])

method = server[1].upper()

configfile = Args.configfile

print\_info("URL : %s" % (Fore.CYAN + server[0]), FileHandle)

print\_info("Method : %s" % (Fore.CYAN + method), FileHandle)

print\_info("Endpoint : %s" % (Fore.CYAN + endpoint), FileHandle)

print\_info("Configfile : %s" % (Fore.CYAN + configfile), FileHandle)

print\_info("Timeout : %s" % (Fore.CYAN + str(float(Args.timeout)) + Fore.MAGENTA + " seconds"), FileHandle)

sm = Desyncr(configfile, host, port, url=server[0], method=method, endpoint=endpoint, SSLFlag=SSLFlagval,

logh=FileHandle, smargs=Args)

sm.run()

if FileHandle is not None:

FileHandle.close()

**APPENDIX A2**

**SCREENSHOTS**



Figure A 1.1 Starting a Virtual Environment Using Docker

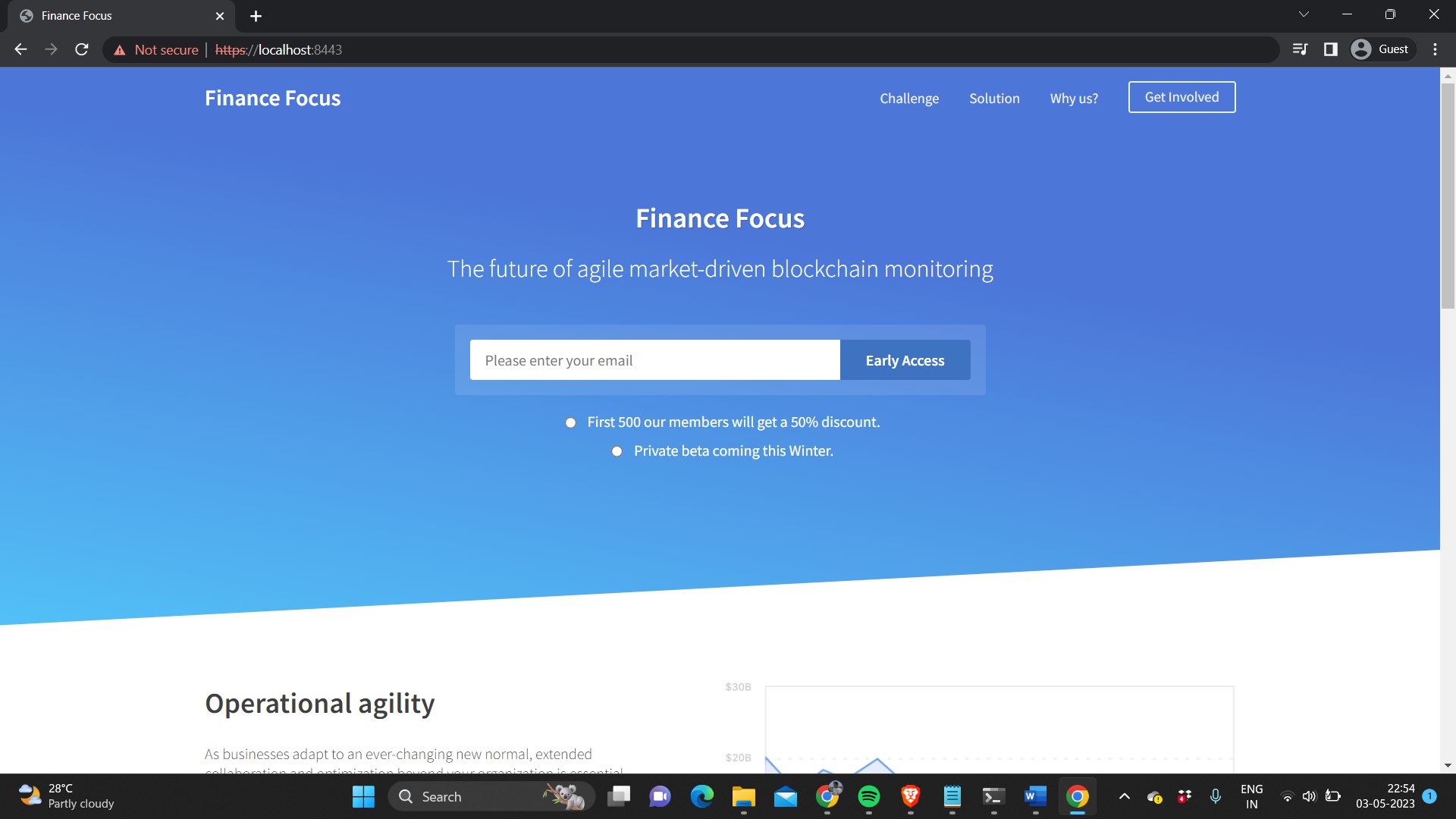


Figure A 1.2 Running a website on localhost port 8443

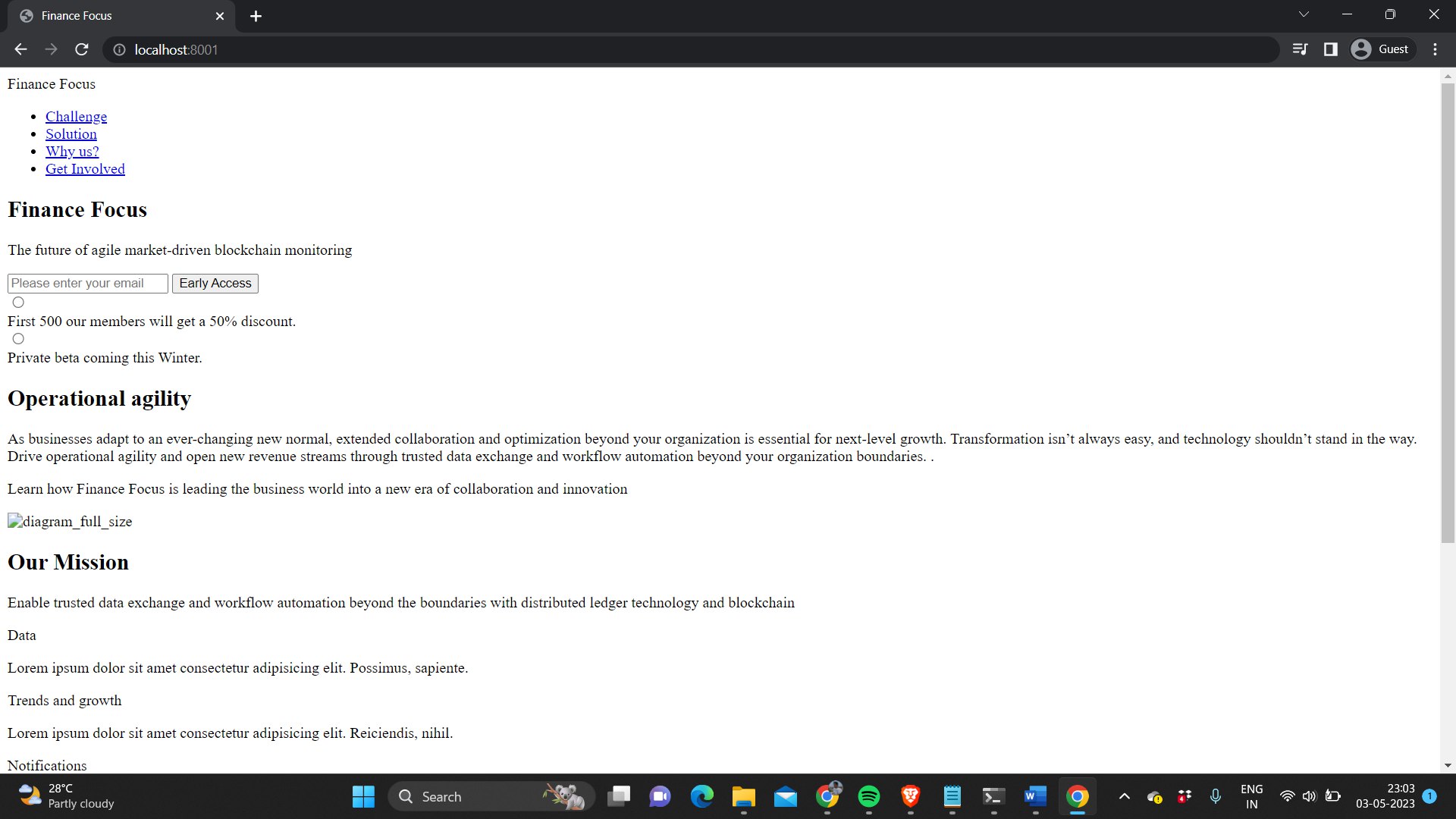


Figure A 1.3: Docker Image Hosting HTML Content of Index Page on Port 8001

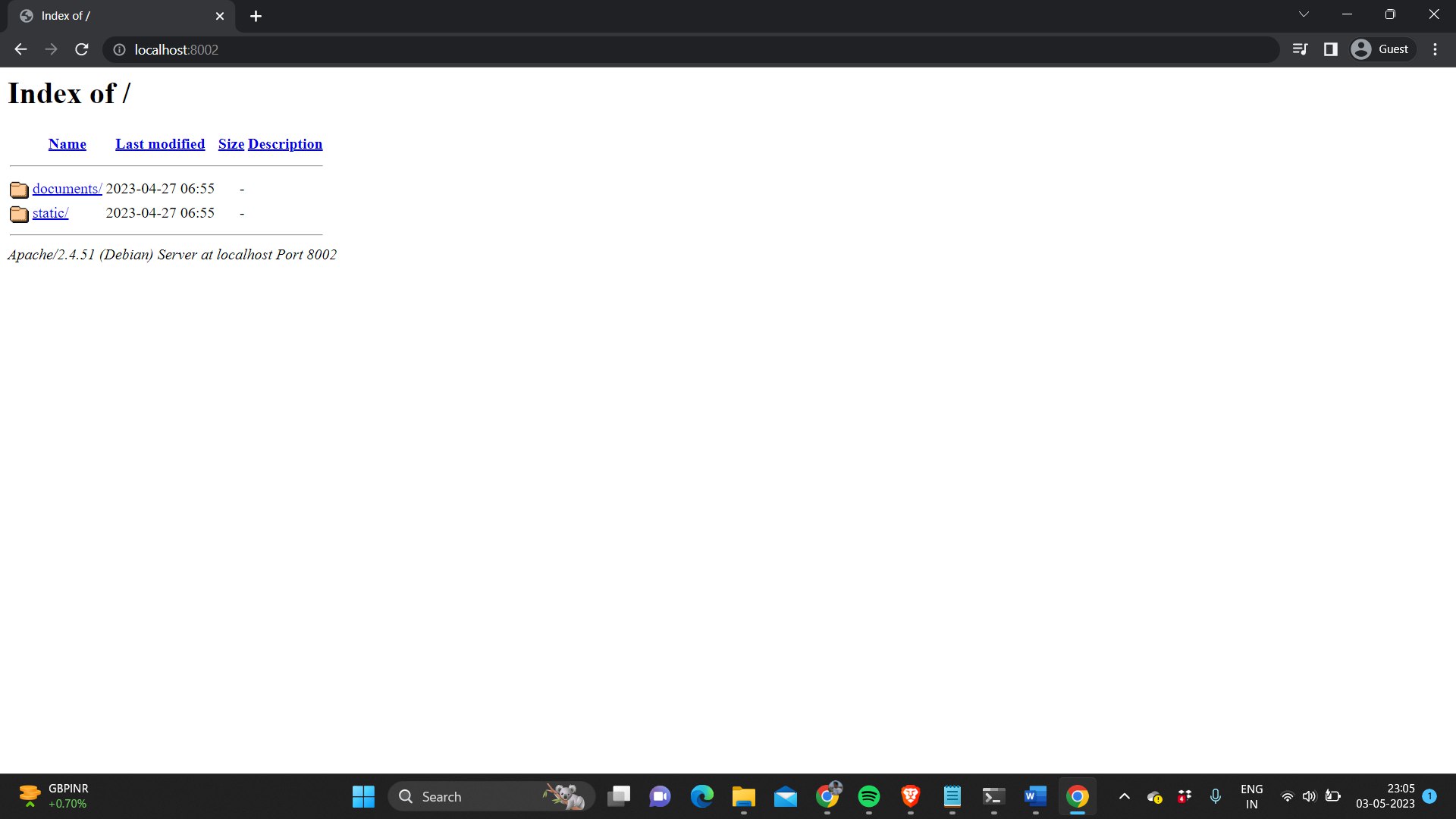


Figure A 1.4: Hosting Documents and Static Files on Port 8002 Using Docker

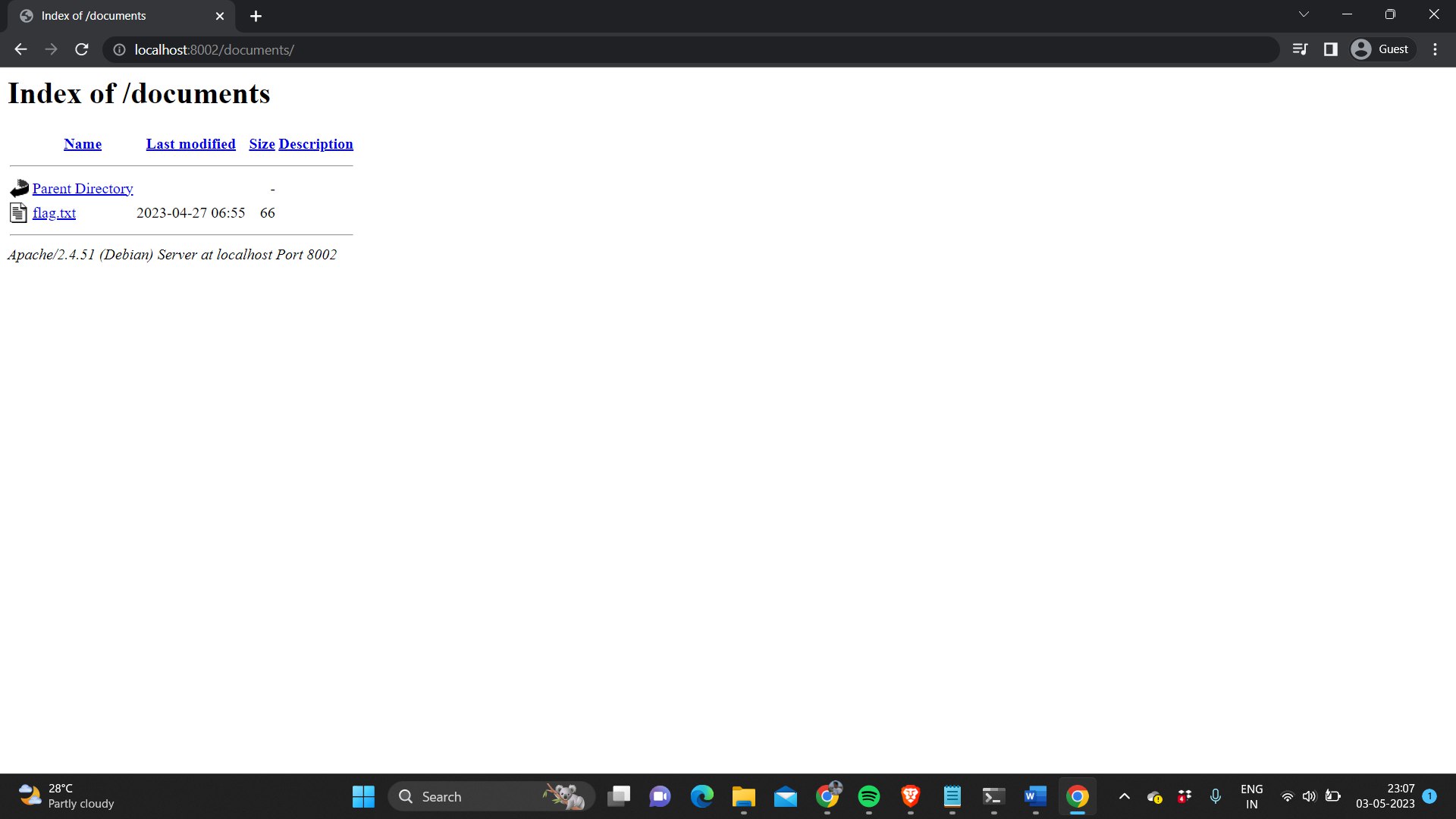


Figure A 1.5: Index of / Documents Directory on Port 8002 Hosting flag.txt

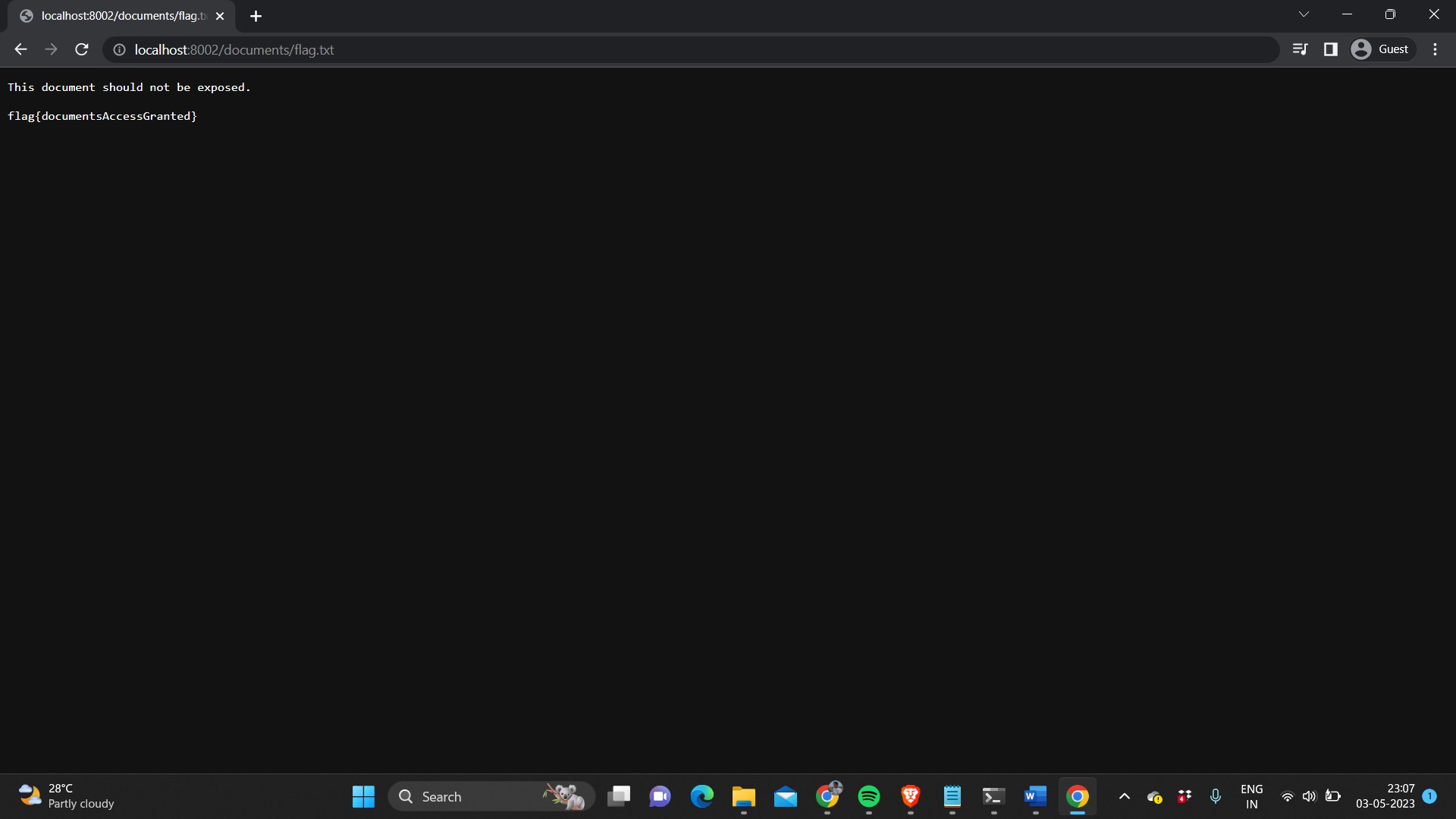


Figure A 1.6 flag.txt file

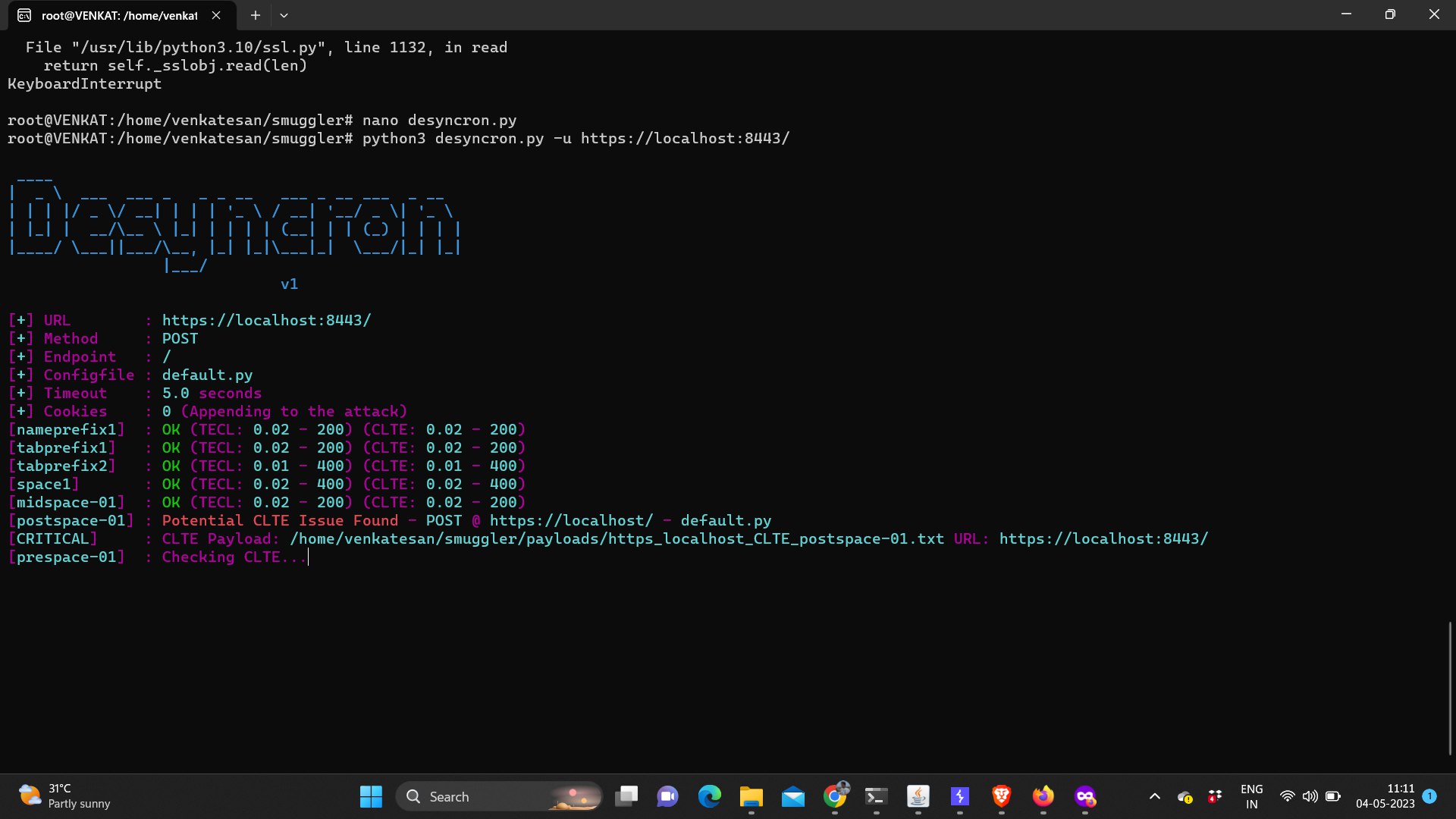


Figure A 1.7 Desyncron Tool

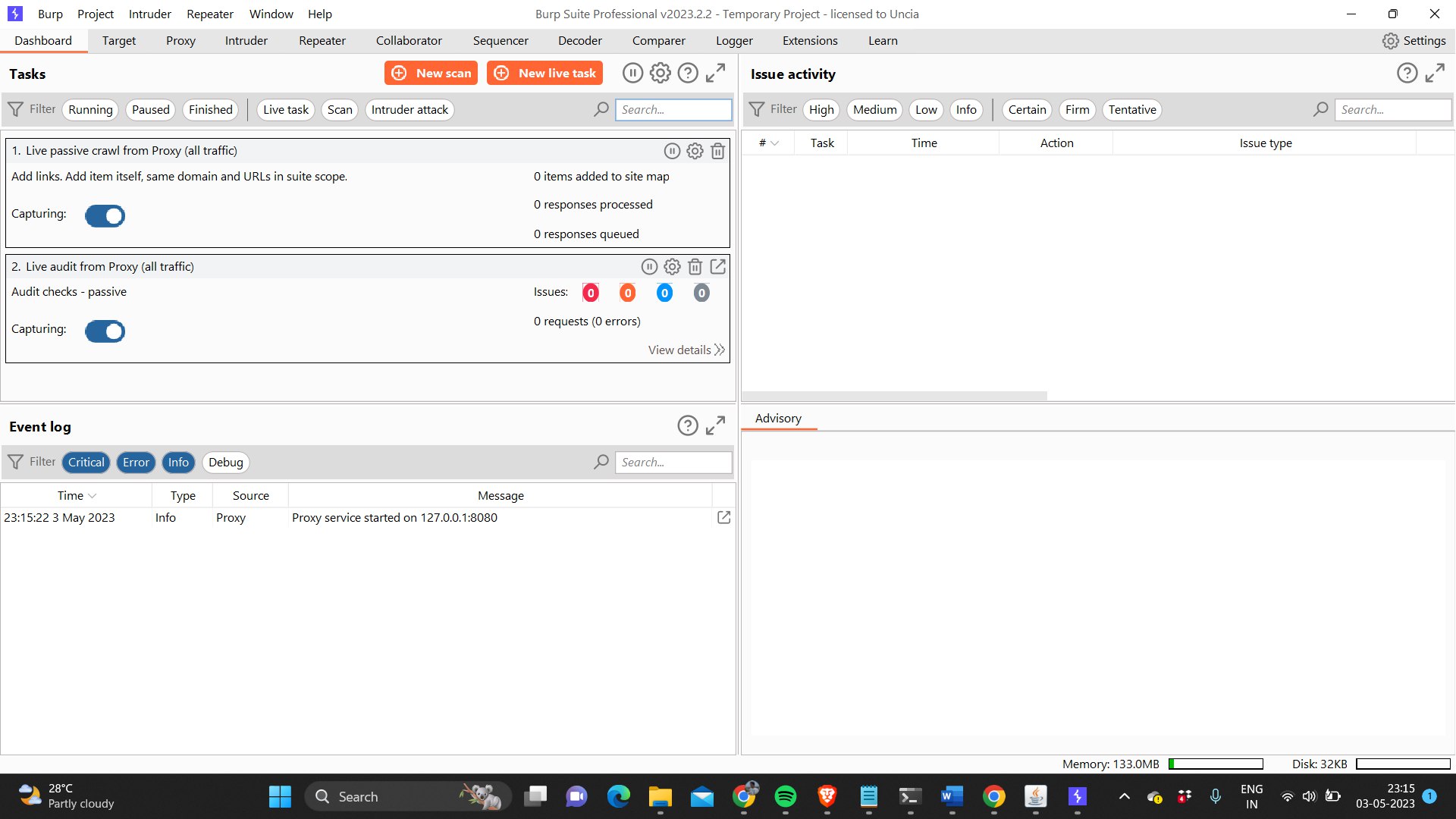


Figure A 1.8: Starting BurpSuite Proxy Service on Localhost Port 8080

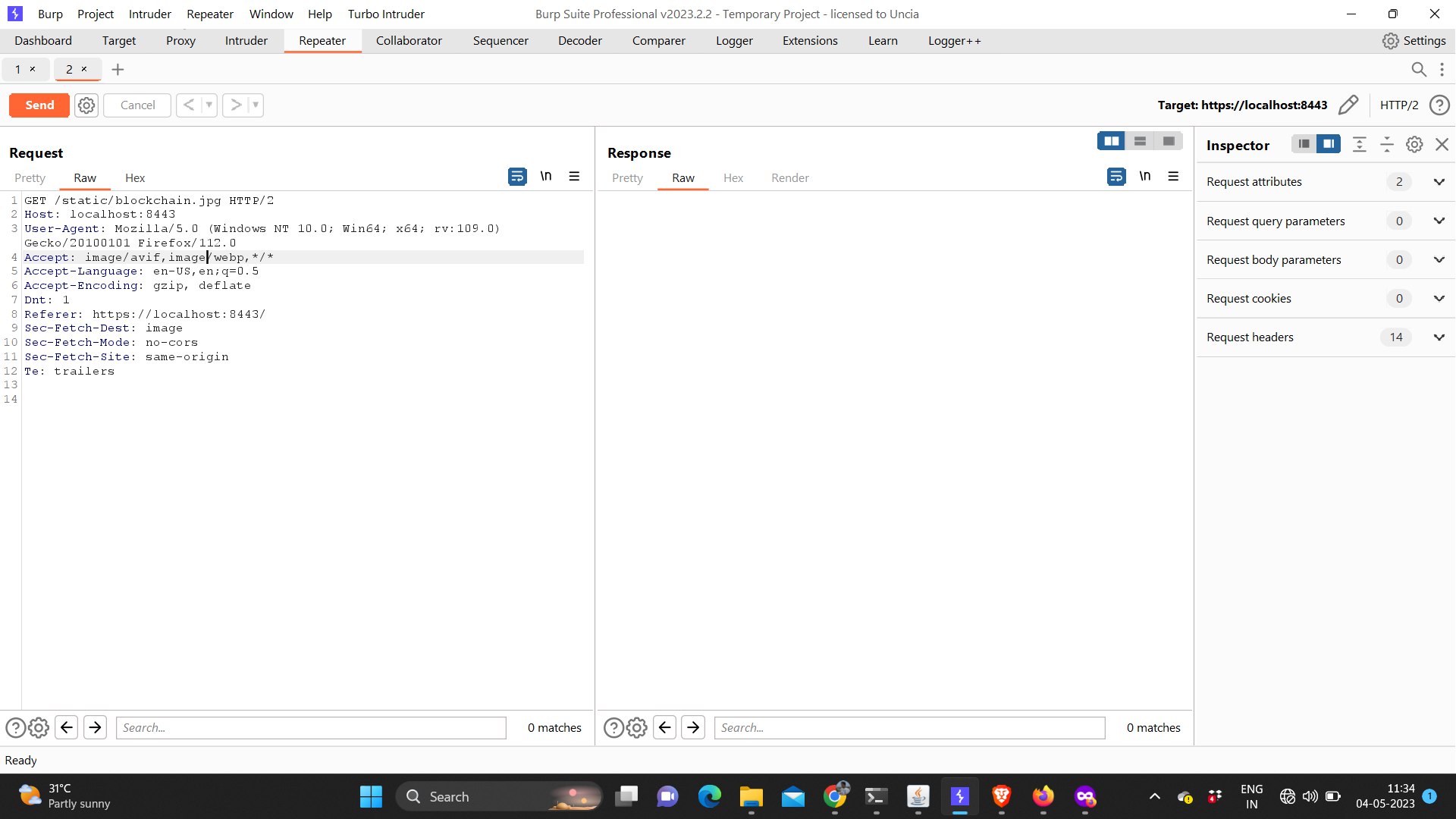
****

Figure A 1.9 Request captured by BurpSuite

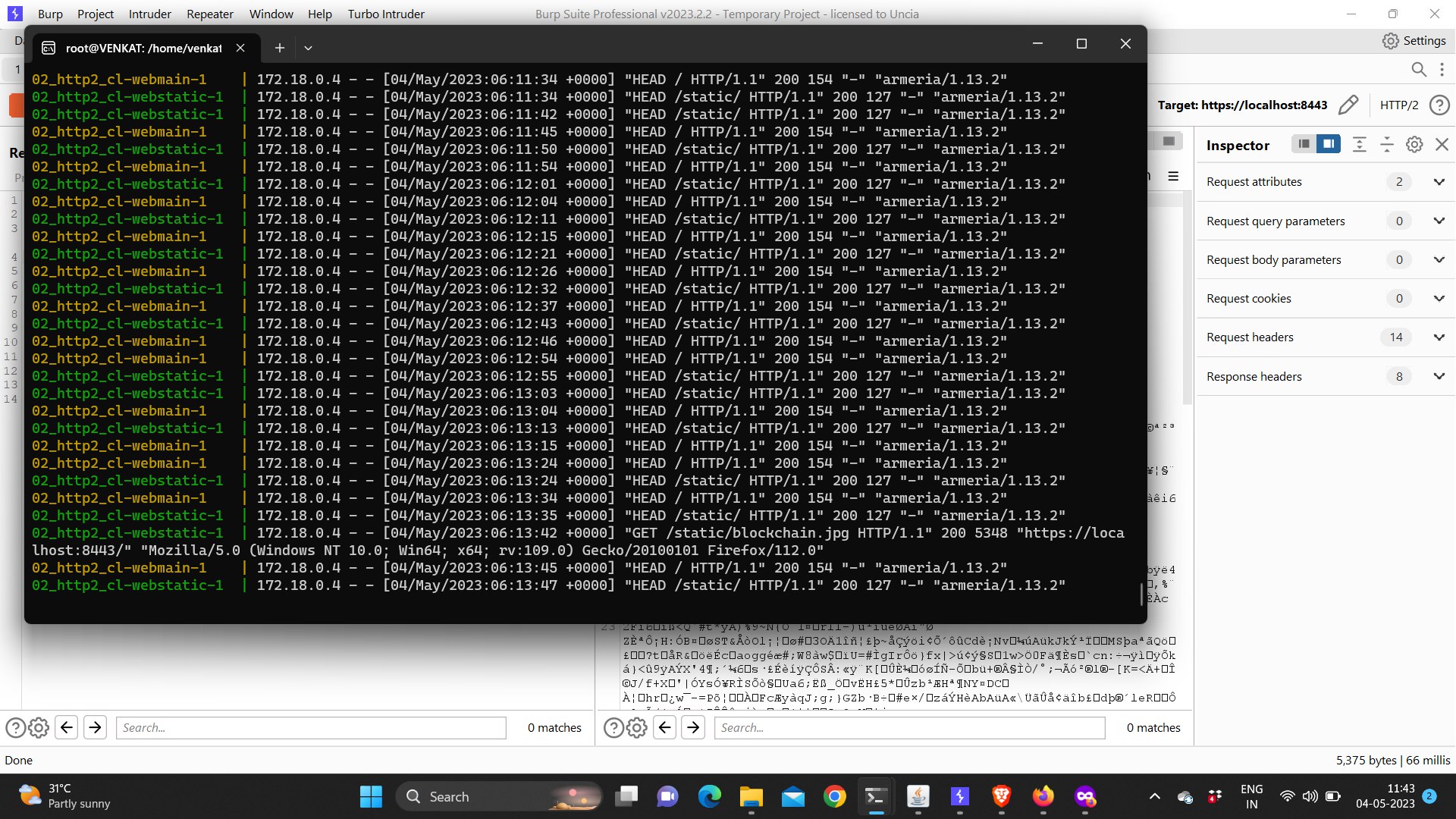
****

Figure A 1.10: Docker Log Displaying Response of Request Sent Through BurpSuite

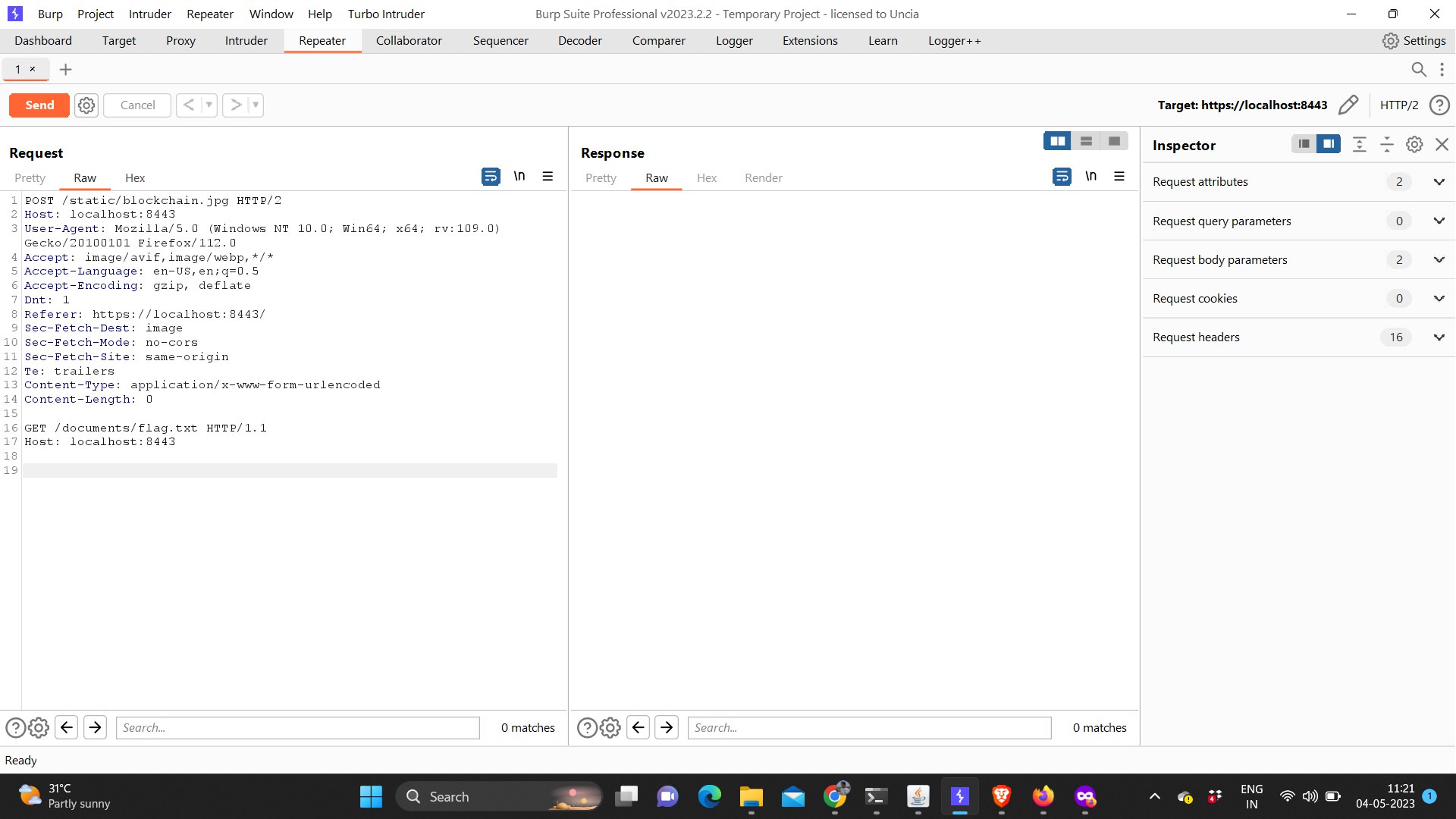
****

Figure A 1.11 Smuggled request

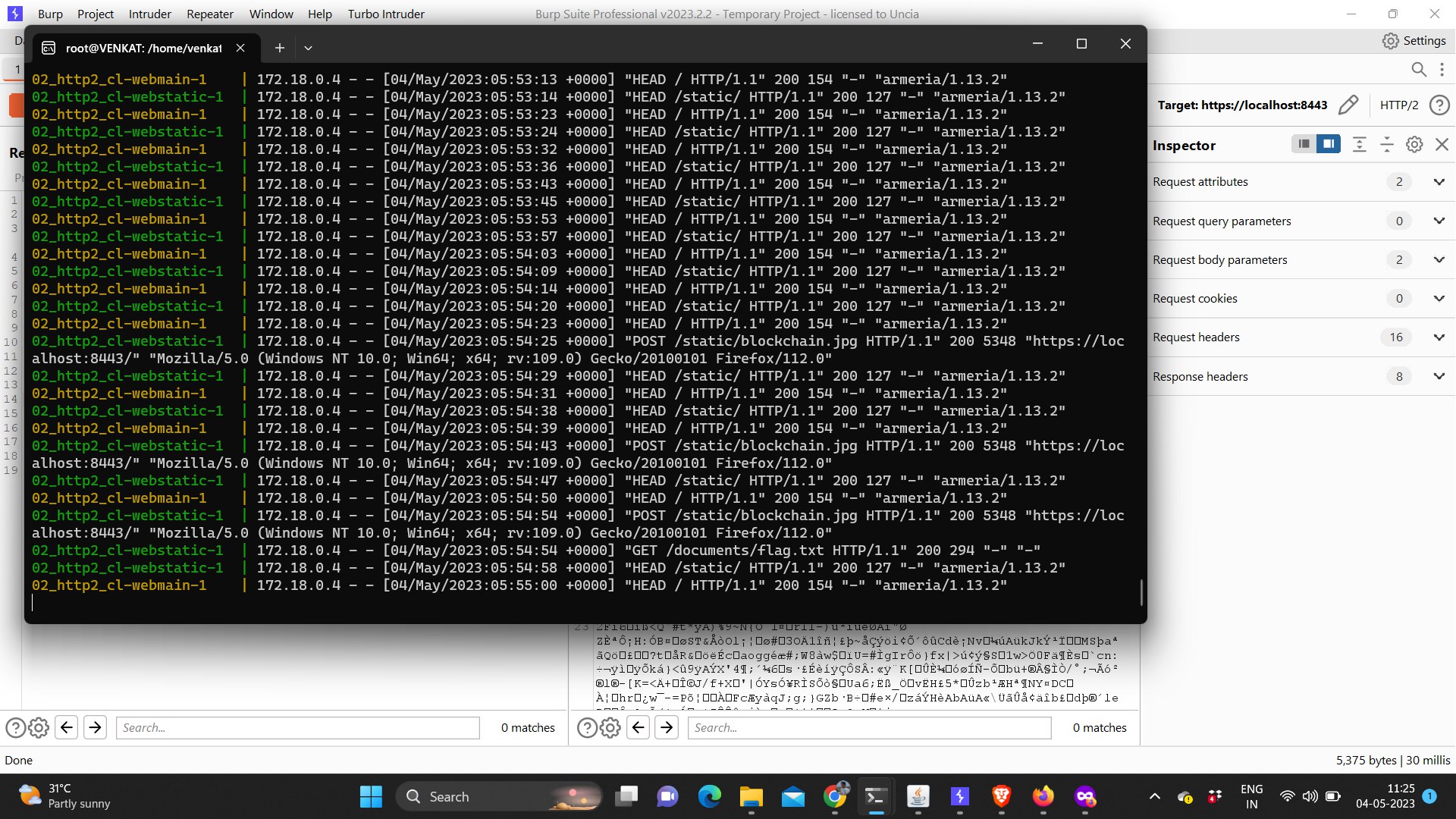
****

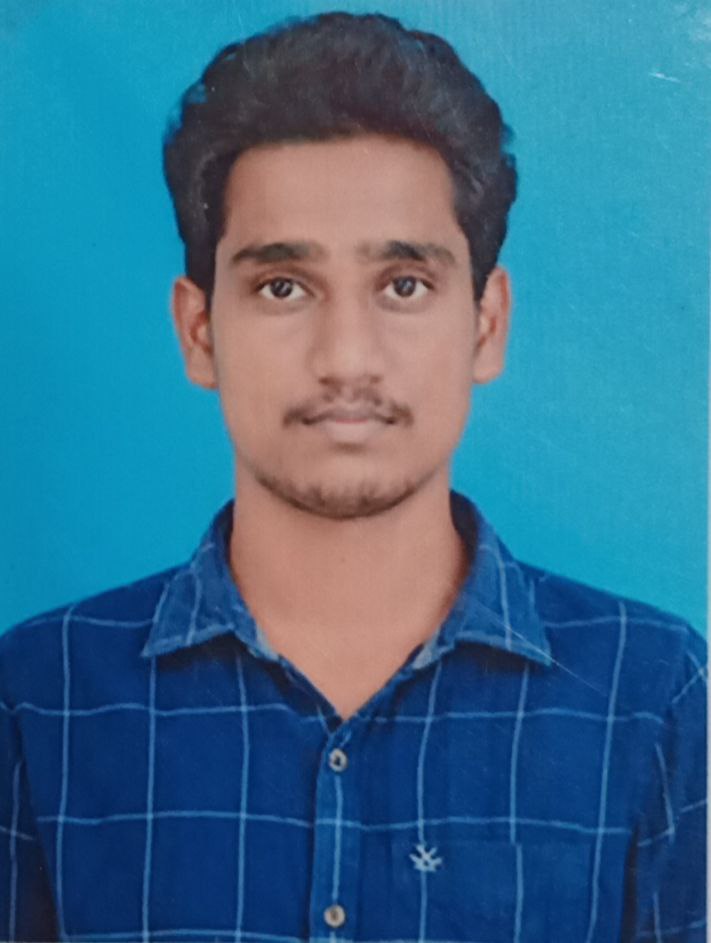
Figure A 1.12: Docker Log Displaying Response of Normal Request and Smuggled Request

**CHAPTER 9**

**TECHNICAL BIOGRAPHY**



Mr. Sujith Kumar is a final year student at B.S. Abdur Rahman Crescent Institute of Science & Technology, Tamil Nadu, India. He is currently pursuing his B.Tech Degree in Computer Science and Engineering and has a strong background in programming languages such as Python and PyScript, as well as experience with statistical analysis tools like SAS and Tableau. Mr. Sujith Kumar has completed certification course on Machine Learning from Kaggle.



Mr. Venkatesan is a final year student at B.S. Abdur Rahman Crescent Institute of Science & Technology in Tamil Nadu, India, where he is pursuing a B.Tech Degree in Computer Science and Engineering. He has over 1 year of experience working as a security researcher at confection.io and more than 2 years of experience as an independent security researcher on HackerOne. Mr. Venkat is a skilled freelancer and ethical hacker with expertise in identifying vulnerabilities in computer systems using programming languages such as Python, C++, and Java. He has extensive experience in penetration testing, bug bounty, threat modeling, and incident response, and has consistently been recognized on the HackerOne leaderboards for his outstanding work in helping clients identify and report security vulnerabilities.

**PLAGARISM REPORT**

