DETECTING AND PREVENTING HTTP DESYNC ATTACKS IN WEB SERVER

S Revathi
Professor
Department of CSE
B.S. Abdur Rahman University
Chennai, India
revathi@crescent.education

Sujith kumar .K
Student
Department of CSE
B.S. Abdur Rahman University
Chennai, India
sujithkumar.cse_c2019@crescent.education

Venkatesan. P
Student
Department of CSE
B.S. Abdur Rahman University
Chennai, India
venkatesanp.cse_c2019@crescent.education

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how malicious actors can send carefully crafted HTTP requests, inducing the client and server to revert to the outdated protocol.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.

Keywords-- HTTP desync attack, request smuggling, client-server desynchronization, web security vulnerability.

I. INTRODUCTION

In recent years, cybersecurity has become more embedded in our everyday lives due to the increasing digitization of our activities and reliance on the internet to access information. However, this also means that our personal information online is vulnerable to cyber threats, as hackers and bug bounty hunters continuously discover new bugs and vulnerabilities in the websites we visit, potentially giving them access to our sensitive data.

Numerous types of weaknesses can be identified on the internet, and the Open Web Application Security Project (OWASP) has a project dedicated to monitoring the "Top 10 Web Application Security Risks." These risks include SQL injection and cross-site scripting (XSS), and extensive research has been conducted in both defensive and offensive measures in these areas. However, several lesser-known vulnerabilities exist, one of which is HTTP Request Smuggling (HRS).

One type of attack in HTTP request smuggling is HTTP desync attacks, where attackers send specially crafted HTTP requests that are interpreted differently by different components of the web server or proxy, causing the server to behave in unexpected ways. This can lead to various security issues and bypass security controls like firewalls, intrusion detection and prevention systems, and web application firewalls. Attackers can use HTTP desync attacks to inject malicious content into legitimate HTTP traffic, steal sensitive data, or even take control of vulnerable web servers.

These attacks can be challenging to detect and mitigate, as they often rely on subtle variations in the HTTP requests that can be difficult to spot. However, there are various techniques and tools available to help prevent and detect HTTP desync attacks, such as updating web server and proxy software to the latest version, implementing strict HTTP request parsing rules, and using specialized security tools to monitor and analyze web traffic.

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

• Request Smuggling: This 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: This attack involves injecting a new line character or other special characters in the HTTP response headers. This can result in the web server or proxy perceiving the response as two distinct responses, which can give rise to different security vulnerabilities.

• Chunked Encoding: This type of HTTP desync attack exploits the way web servers 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 .

• 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.

• 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.

• Cache poisoning: Cache poisoning in HTTP request smuggling is an attack type that exploits vulnerabilities in caching systems to manipulate the caching data in browsers, 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. This can enable the attacker to steal sensitive information or perform unauthorized activities on behalf of the client.

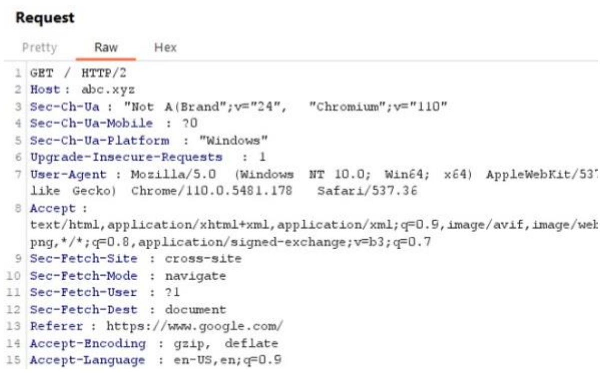
These are just a few examples of the techniques that attackers can use it to exploit the vulnerability. As with any type of cyber-attack, attackers are constantly evolving their tactics, so it's essential to stay up-to-date with the latest security updates and implement best practices to protect against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

II. HTTP/2

Prior to discussing the mechanics of server operation and request transmission, it is important to establish a foundational understanding of HTTP/2.

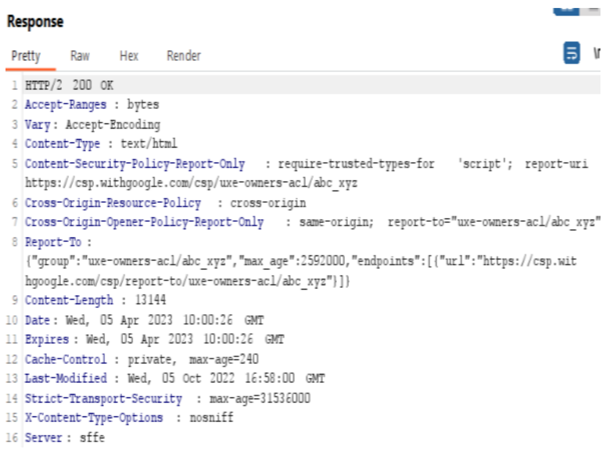
In HTTP/2, a single TCP connection is used to enable concurrent transmission of multiple requests and responses, leading to more efficient use of network resources. This means that a client can initiate several requests over the same TCP connection without having to wait for each response before sending the next request. Similarly, a server can send multiple responses over the same connection and interleave them with the requests. This is achieved through the use of frames and streams, which enable the connection to be divided into independent channels. The use of parallel processing of multiple requests and responses minimizes the latency and overhead related to creating and terminating multiple TCP connections. HTTP/2 delivers significant improvements in performance over HTTP/1.0 and 1.1, thanks to its support for multiplexing, header compression, and other optimizations.

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.



```
GET / HTTP/2
Host : abc.xyz
Sec-Ch-Ua : "Not A(Brand";v="24",  "Chromium";v="110"
Sec-Ch-Ua-Mobile : ?0
Sec-Ch-Ua-Platform : "Windows"
Upgrade-Insecure-Requests : 1
User-Agent : Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/53
like Gecko) Chrome/110.0.5481.178  Safari/537.36
Accept :
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/we
png,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site : cross-site
Sec-Fetch-Mode : navigate
Sec-Fetch-User : ?1
Sec-Fetch-Dest : document
Referer : https://www.google.com/
Accept-Encoding : gzip, deflate
Accept-Language : en-US,en;q=0.9
```

Fig. 1. Sending a request to www.abc.xyz



```
HTTP/2 200 OK
Accept-Ranges : bytes
Vary : Accept-Encoding
Content-Type : text/html
Content-Security-Policy-Report-Only  : require-trusted-types-for  'script';  report-uri
https://csp.withgoogle.com/csp/uxe-owners-acl/abc_xyz
Cross-Origin-Resource-Policy  : cross-origin
Cross-Origin-Opener-Policy-Report-Only  : same-origin;  report-to="uxe-owners-acl/abc_xyz"
Report-To :
{"group":"uxe-owners-acl/abc_xyz","max_age":2592000,"endpoints":[{"url":"https://csp.wit
hgoogle.com/csp/report-to/uxe-owners-acl/abc_xyz"}]}
Content-Length : 13144
Date : Wed, 05 Apr 2023 10:00:26 GMT
Expires : Wed, 05 Apr 2023 10:00:26 GMT
Cache-Control : private, max-age=240
Last-Modified : Wed, 05 Oct 2022 16:58:00 GMT
Strict-Transport-Security  : max-age=31536000
X-Content-Type-Options  : nosniff
Server : sffe
```

Fig. 2. Respective response for above sent request

Usually, it is in the HTTP header the type of request is defined. HTTP provides two commonly used request methods, GET and POST, which are used to retrieve and send data from/to a server respectively.

The GET method is utilized to ask for a resource from a server. Once the client makes a GET request, the server obtains the resource specified in the request URL and returns it to the client in the response. The data sent along with a GET request is added to the URL query string, which is viewable in the browser's address bar. This method is considered safe as it does not change any data on the server.

On the contrary, the POST method is used to transmit data to a server to create or modify a resource. In a POST request, the data is included in the request body rather than the URL, enabling the transfer of large amounts of data such as form submissions. Unlike the GET method, the POST method is not safe as it can modify data on the server.

To summarize, the GET method is employed to retrieve data from a server, whereas the POST method is utilized to transmit data to a server to create or modify a resource. The appropriate method should be selected depending on the intended use case to guarantee that data is managed safely and effectively.

HOW A SERVER WORKS

In the context of HTTP/2, the front-end and back-end of a server collaborate to provide an optimized and efficient web experience to the end-user. The front-end of a server is responsible for managing incoming HTTP/2 requests from clients and controlling the associated network connections. On the other hand, the back-end is responsible for processing these requests, producing dynamic content, and accessing resources such as databases.

With HTTP/2, the front-end can effectively multiplex multiple requests and responses over a single TCP connection, which permits concurrent processing of requests, thus minimizing latency. This is particularly useful for web applications that require multiple resources or real-time updates.

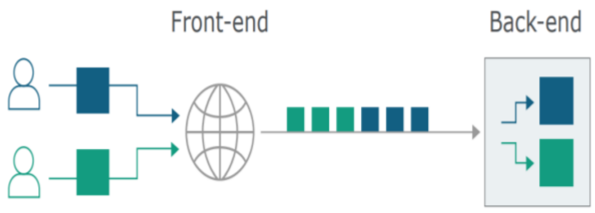


Fig. 3. Showing the working of a server

Fig. 3 depicts the basic working 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.

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. HTTP/2 provides header compression and other optimizations that the back-end of a server can leverage to improve performance. For instance, header compression drastically reduces the size of HTTP/2 headers, enabling faster transmission and processing of requests.

HTTP DESYNC ATTACK

HTTP Desync Attacks are a type of advanced web application attack that takes advantage of the differences in how HTTP/1.x and HTTP/2 servers handle HTTP requests with unexpected or invalid headers. These attacks can be used to bypass security mechanisms, gain access to sensitive information, or execute arbitrary code on the server. HTTP Desync Attacks are highly sophisticated and require a deep understanding of web server behavior and the HTTP protocol.

HTTP/1.x is an older version of the HTTP protocol that is still widely used today. In HTTP/1.x, servers process incoming HTTP requests sequentially. That is, they wait for each request to be fully received and processed before moving on to the next request. This can create inconsistencies in how servers interpret and process HTTP requests, which attackers can exploit to carry out HTTP Desync Attacks.

Modern web applications often rely on HTTP pipelining, which allows clients to send multiple requests without waiting for the server's response. This can create additional inconsistencies in how servers interpret HTTP requests, which attackers can use to their advantage. By sending specially crafted HTTP requests with unexpected or invalid headers, attackers can trick the server into interpreting the request differently from how the client intended.

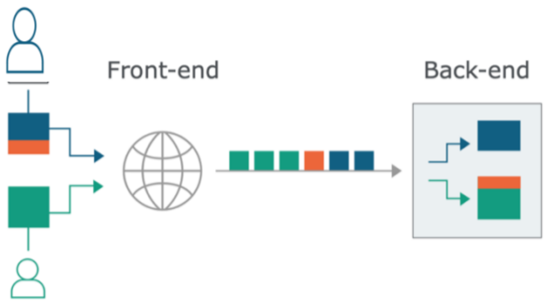


Fig. 4. HTTP Desync Attack Mechanism

HTTP/2 is a newer version of the HTTP protocol that was designed to address some of the limitations of HTTP/1.x. In HTTP/2, servers use a binary protocol that allows for multiplexing, which enables the server to process multiple requests in parallel. However, this can also create discrepancies in how the server interprets HTTP requests, which attackers can exploit to carry out HTTP Desync Attacks.

Attackers can use various techniques to carry out HTTP Desync Attacks on both HTTP/1.x and HTTP/2 servers. One technique is to manipulate the content-length header, which specifies the length of the request body in bytes. By sending a request with an incorrect content-length header, an attacker can trick the server into interpreting the request differently. Another technique is to send a chunked request with a zero-length chunk, which can cause the server to interpret the request as multiple requests. Attackers can also use a mix of transfer-encoding and content-length headers to create discrepancies in how the server processes the request.

HTTP Desync Attacks are highly sophisticated and require a good understanding of web server behavior and the HTTP protocol. As such, they are not easy to carry out, and most attackers do not have the necessary expertise to execute these attacks successfully. However, as web applications become more complex, and the number of HTTP requests that servers handle continues to increase, HTTP Desync Attacks are becoming a more significant threat to web application security.

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.

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 5 illustrates the underlying system architecture of how the request is sent and is responded respectively.

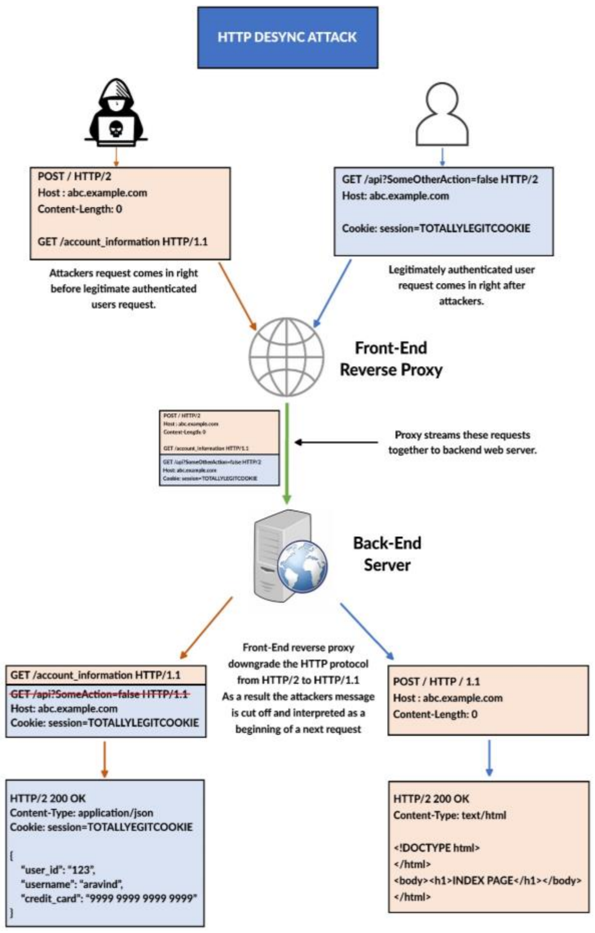

HTTP DESYNC ATTACK
POST / HTTP/2
Host : abc.example.com
Content-Length: 0
GET /account_information HTTP/1.1
GET /api?SomeOtherAction=false HTTP/2
Host: abc.example.com
Cookie: session=TOTALLYLEGITCOOKIE
Attackers request comes in right before legitimate authenticated users request.
Legitimately authenticated user request comes in right after attackers.
Front-End Reverse Proxy
Proxy streams these requests together to backend web server.
Back-End Server
Front-End reverse proxy downgrade the HTTP protocol from HTTP/2 to HTTP/1.1 As a result the attackers message is cut off and interpreted as a beginning of a next request
GET /account_information HTTP/1.1
~~GET /api?SomeAction=false HTTP/1.1~~
Host: abc.example.com
Cookie: session=TOTALLYLEGITCOOKIE
POST / HTTP / 1.1
Host : abc.example.com
Content-Length: 0
HTTP/2 200 OK
Content-Type: application/json
Cookie: session=TOTALLYEGITCOOKIE
{ "user_id": "123", "username": "aravind", "credit_card": "9999 9999 9999 9999" }
HTTP/2 200 OK
Content-Type: text/html
<!DOCTYPE html>
</html>
<body><h1>INDEX PAGE</h1></body>
</html>

Fig. 5. System Architecture

The system architecture described in Figure 5 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 5 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.

Flaws in HTTP/2 protocol

HTTP/2 is a recent version of the HTTP protocol, aiming to improve web application performance by decreasing latency and enhancing network efficiency. However, HTTP/2 is not immune to HTTP desync attacks, and there are several protocol flaws that attackers can exploit.

One of the most significant weaknesses in the HTTP/2 protocol is the way it handles request and response messages. In contrast to HTTP/1.x, where every request and response message is treated as a separate HTTP transaction, HTTP/2 permits multiple request and response messages to be multiplexed over a single TCP connection. This multiplexing feature can lead to ambiguity in the order of HTTP messages, resulting in HTTP desync attacks.

Another issue with the HTTP/2 protocol is related to the way it manages header compression. HTTP/2 employs a header compression technique known as HPACK, which compresses HTTP headers to reduce the size of requests and responses. However, the use of HPACK can result in uncertainty about the length of HTTP messages, which attackers can exploit for carrying out HTTP desync attacks.

Moreover, the HTTP/2 protocol does not have clear instructions for parsing and processing HTTP requests and responses, which attackers can take advantage of to manipulate requests and responses, leading to HTTP desync attacks.

To prevent HTTP desync attacks in the HTTP/2 protocol, web application developers should implement secure coding practices, comply with strict HTTP/2 protocol compliance rules, and use web application firewalls capable of detecting and preventing HTTP desync attacks. Additionally, web server and proxy administrators should configure their systems to adhere to strict HTTP/2 protocol compliance rules and ensure that their systems are up to date with the latest security patches.

DOWNGRADING FROM HTTP/2 TO HTTP/1.1: A FLAW WITH SECURITY IMPLICATIONS

Downgrading from HTTP/2 to HTTP/1.1 can introduce security vulnerabilities that are not present in HTTP/2, which offers several security features such as header compression, server push, and binary framing that are not available in HTTP/1.1.

When a server responds to an HTTP/2 request with an HTTP/1.1 message, it can leave the client vulnerable to attacks that rely on these missing security features. Attackers can exploit this vulnerability to carry out HTTP request smuggling attacks, as discussed earlier, and bypass security mechanisms.

Additionally, HTTP/1.1 messages are not multiplexed like HTTP/2 messages, and thus, HTTP/1.1 clients have to establish multiple connections to a server to process several requests simultaneously, leading to increased latency and reduced network efficiency. This can cause performance issues.

In summary, downgrading from HTTP/2 to HTTP/1.1 can lead to security vulnerabilities and performance issues. It is generally recommended to use HTTP/2 whenever possible.

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.

FLOW DIAGRAM

The proposed system workflow for detecting and preventing HTTP desync attacks in web servers is shown in the Fig. 4.

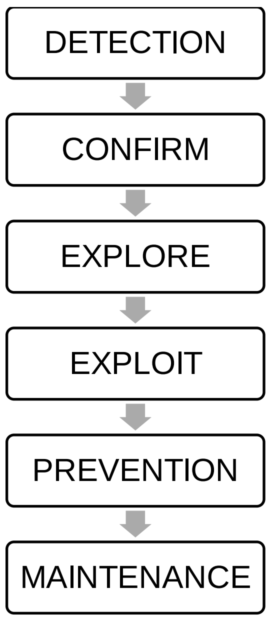


Fig. 6. PROPOSED SYSTEM WORKFLOW

1. 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.

2. CONFIRM

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

3. 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.

4. 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.

5. 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.

6. MAINTENANCE

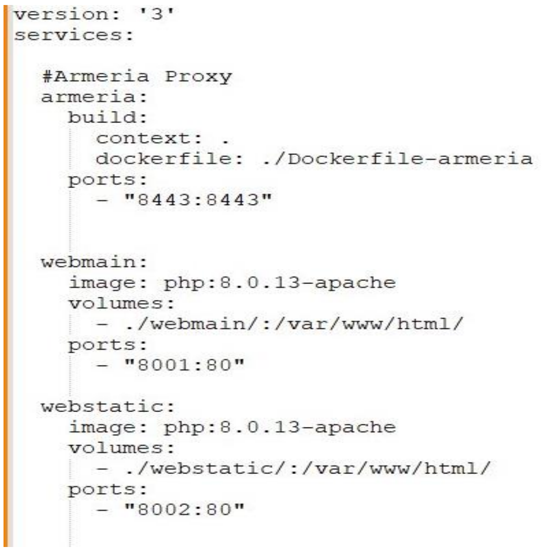
In conclusion, the maintenance stage of the workflow is crucial for ensuring that web applications are continually protected against potential threats. One of the most important aspects of maintenance is to keep the WAF up-to-date with the latest rules and techniques to address emerging threats. As new 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.

METHODOLOGY

A. Setting up the Environment

For the implementation of the HTTP desync attack testing framework, various types of web servers were used as both frontend and backend components. The server is 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.

To set up a consistent testing environment across different systems, Docker containers were utilized, and Docker Compose was employed for container management. The Docker Compose configuration file, shown in Figure 5.1, defines the setup for the frontend, backend, webmain, and webstatic files, which are accessed through ports 8001:80 and 8002:80, respectively.



```
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"
```

Fig. 7. Docker compose.yml

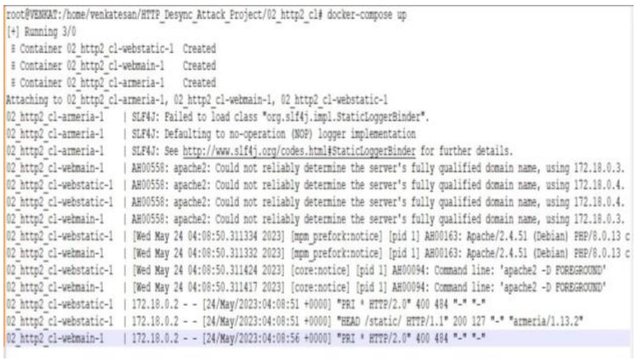
In the implementation phase, the chosen frontend component is Armeria Proxy, which is an open-source proxy server. Armeria Proxy offers a range of advanced features for efficient handling and routing of network traffic, enabling seamless communication between clients and servers.

For the backend component, the Apache HTTP Server is employed. Apache is a highly popular and reliable web server known for its support of multiple programming languages and frameworks. It boasts an extensive feature set, including comprehensive HTTP/HTTPS support, virtual hosting capabilities, URL rewriting, and extensive module support. Given these features, Apache is an excellent choice for hosting the backend of the project.

Steps for creating the Docker environment:

- Install Docker and Docker Compose on the target system(s), ensuring that the versions are compatible with the project requirements.
- Create a directory for your project and navigate to it using the command line or terminal.
- Create a new file named docker-compose.yml and open it in a text editor.
- Copy the contents of Figure 7, the example Docker Compose configuration file, into docker-compose.yml.
- Save the file and close the text editor.
- Place the frontend and backend files in the appropriate directories as specified in the Docker Compose configuration.
- Ensure that the file paths and names match the configuration.
- Open a command line or terminal and navigate to the directory containing the docker-compose.yml file.
- Execute the command docker-compose up to start the containers defined in the configuration file.
- Docker Compose will automatically pull necessary container images as shown in Figure 8, if they are not already present on the system.
- Monitor the output to ensure that the containers are successfully started.
- Any error messages or issues should be addressed accordingly.
- Once the containers are running, you can access the frontend application through the specified port, such as localhost:8001, in a web browser.
- Similarly, the backend application can be accessed through the specified port, such as localhost:8002.

In Figure 8, the log of the Docker Compose up command starting the containers is displayed. This log provides valuable information about the initialization and execution of the containers.



```
root@VENKAT:/home/venkatesan/HTTP_Desync_Attack_Project/02_http2_cl# docker-compose up
[+] Running 3/0
 ⠿ Container 02_http2_cl-webstatic-1  Created
 ⠿ Container 02_http2_cl-webmain-1    Created
 ⠿ Container 02_http2_cl-armeria-1    Created
Attaching to 02_http2_cl-armeria-1, 02_http2_cl-webmain-1, 02_http2_cl-webstatic-1
02_http2_cl-armeria-1    | SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
02_http2_cl-armeria-1    | SLF4J: Defaulting to no-operation (NOP) logger implementation
02_http2_cl-armeria-1    | SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
02_http2_cl-webmain-1    | AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 172.18.0.3.
02_http2_cl-webstatic-1  | AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 172.18.0.4.
02_http2_cl-webstatic-1  | AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 172.18.0.4.
02_http2_cl-webmain-1    | AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 172.18.0.3.
02_http2_cl-webstatic-1  | [Wed May 24 04:08:50.311334 2023] [mpm_prefork:notice] [pid 1] AH00163: Apache/2.4.51 (Debian) PHP/8.0.13 c
02_http2_cl-webmain-1    | [Wed May 24 04:08:50.311332 2023] [mpm_prefork:notice] [pid 1] AH00163: Apache/2.4.51 (Debian) PHP/8.0.13 c
02_http2_cl-webstatic-1  | [Wed May 24 04:08:50.311424 2023] [core:notice] [pid 1] AH00094: Command line: 'apache2 -D FOREGROUND'
02_http2_cl-webmain-1    | [Wed May 24 04:08:50.311417 2023] [core:notice] [pid 1] AH00094: Command line: 'apache2 -D FOREGROUND'
02_http2_cl-webstatic-1  | 172.18.0.2 - - [24/May/2023:04:08:51 +0000] "PRI * HTTP/2.0" 400 484 "-" "-"
02_http2_cl-webstatic-1  | 172.18.0.2 - - [24/May/2023:04:08:51 +0000] "HEAD /static/ HTTP/1.1" 200 127 "-" "armeria/1.13.2"
02_http2_cl-webmain-1    | 172.18.0.2 - - [24/May/2023:04:08:56 +0000] "PRI * HTTP/2.0" 400 484 "-" "-"
```

Fig. 8. Docker compose up

By analyzing the log, you can track the progress of each container as it starts and ensure that the initialization process completes successfully. It typically includes details such as container names, status, and any error messages or warnings that may arise during startup.

After the containers have successfully started, you can access the frontend files by navigating to localhost:8443 in a web browser. Similarly, the backend files can be accessed by visiting localhost:8002.

A simple webpage is displayed in Figure 9 on port 8443, which is a usual website that is created having frontend and backend components.

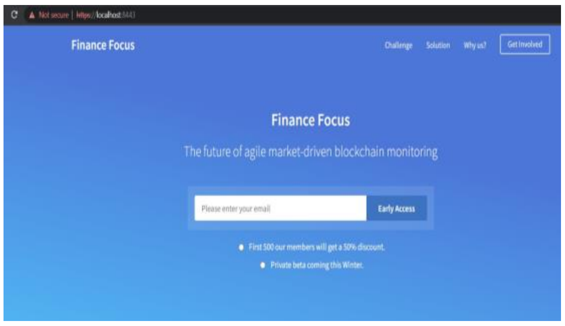
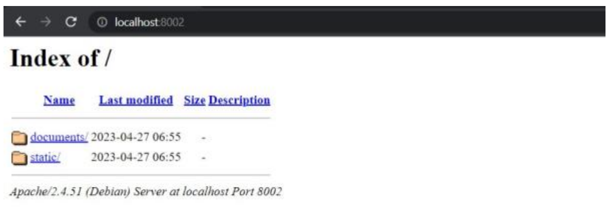

Not secure | https://localhost:8443
Finance Focus
Challenge
Solution
Why us?
Get involved
Finance Focus
The future of agile market-driven blockchain monitoring
Please enter your email
Early Access
First 500 our members will get a 50% discount.
Private beta coming this Winter.

Fig. 9. Running a website on port 8443

Figure 10 showcases the implementation of hosting documents and static files on Port 8002 using Docker. The figure depicts the index page, which consists of two directories: "documents" and "static."

The "documents" directory serves as a storage location for various types of documents, such as PDF files, Word documents, or text files. These documents can be accessed and retrieved by clients through the specified port and directory path.

On the other hand, the "static" directory contains static files, which typically include assets like images, CSS files, JavaScript files, or other resources that are required for the website's visual presentation and functionality. Clients can access these static files by referencing the appropriate directory and file name within the "static" directory.



localhost:8002

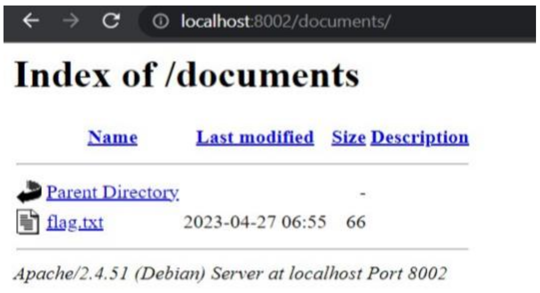
Index of /

Name	Last modified	Size	Description
documents/	2023-04-27 06:55	-	
static/	2023-04-27 06:55	-	

Apache/2.4.51 (Debian) Server at localhost Port 8002

Fig. 10. Hosting Documents and Static Files on Port 8002 Using Docker

Within the "documents" directory, there exists a specific file named flag.txt, which is classified as a forbidden file. Figure 11 highlights this file and indicates that only the admin has the permission to access it. If a non-admin user attempts to access this file, it will result in a "404 File not found" error.



localhost:8002/documents/

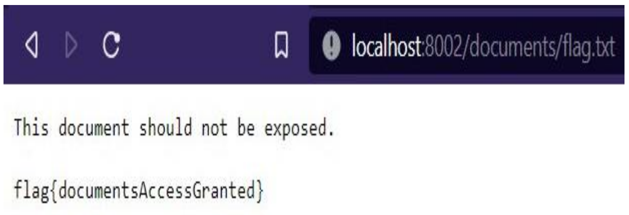
Index of /documents

Name	Last modified	Size	Description
Parent Directory		-	
flag.txt	2023-04-27 06:55	66	

Apache/2.4.51 (Debian) Server at localhost Port 8002

Fig. 11. Index of / Documents Directory on Port 8002 Hosting flag.txt

But if we login as an admin, then the file flag.txt is accessible showing the content portrayed in the Figure 12.



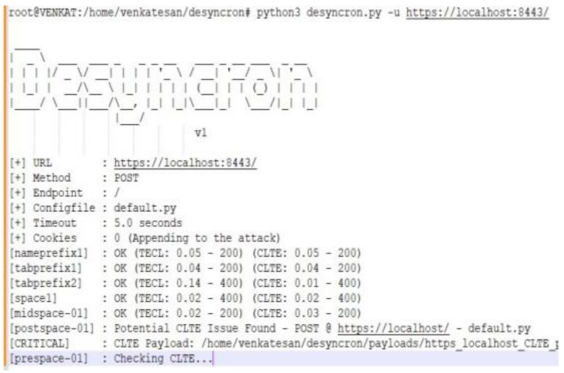
localhost:8002/documents/flag.txt

```
This document should not be exposed.

flag{documentsAccessGranted}
```

Fig. 12. flag.txt file

Next, we are going to test that website for HTTP Desync vulnerability, for this a Python detection tool called Desyncron is used to test the web servers for HTTP desync attacks, Desyncron is an automated tool that systematically tests web servers for vulnerabilities related to HTTP desynchronization. The Command Line Interface of Desyncron tool is illustrated in the Figure 13. It achieves it by sending meticulously crafted HTTP requests to the target server, utilizing various methods and techniques to probe for potential vulnerabilities.



```
root@VENKAT:/home/venkatesan/desyncron# python3 desyncron.py -u https://localhost:8443/

                                   v1

[+] URL        : https://localhost:8443/
[+] Method     : POST
[+] Endpoint   : /
[+] Configfile : default.py
[+] Timeout    : 5.0 seconds
[+] Cookies    : 0 (Appending to the attack)
[nameprefix1]  : OK (TECL: 0.05 - 200) (CLTE: 0.05 - 200)
[tabprefix1]   : OK (TECL: 0.04 - 200) (CLTE: 0.04 - 200)
[tabprefix2]   : OK (TECL: 0.14 - 400) (CLTE: 0.01 - 400)
[space1]       : OK (TECL: 0.02 - 400) (CLTE: 0.02 - 400)
[midspace-01]  : OK (TECL: 0.02 - 200) (CLTE: 0.03 - 200)
[postspace-01] : Potential CLTE Issue Found - POST @ https://localhost/ - default.py
[CRITICAL]     : CLTE Payload: /home/venkatesan/desyncron/payloads/https_localhost_CLTE_
[prespace-01]  : Checking CLTE...
```

Fig. 13. Working of Desyncron Tool

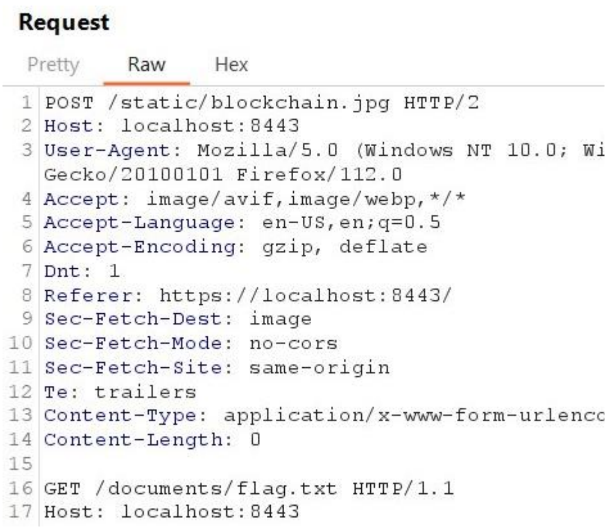
One of the key aspects of Desyncron approach is the careful manipulation of HTTP request parameters. For example, it may send requests with different content lengths, ranging from zero to larger values, in order to assess how the server interprets and handles such variations. By systematically exploring different scenarios, Desyncron aims to identify any irregularities or inconsistencies in the server's response, which could indicate the presence of an HTTP desync vulnerability.

In addition to content length manipulation, Desyncron employs various other techniques to craft its requests. It may modify headers, payloads, or other request elements to trigger potential desynchronization issues within the server. By using the Desyncron tool on localhost 8443, a vulnerability has been found postspace-01 CLTE issue.

The "postspace-01 CLTE" issue refers to a specific technique where an attacker can exploit the way the server interprets the "Content-Length" header in combination with the "Transfer-Encoding: chunked" header. By manipulating the request and inserting additional whitespace or characters after the "Content-Length" value, an attacker may cause the server to misinterpret the request and potentially introduce security vulnerabilities.

Let’s say, that a vulnerability has been found on that website, there could be an error in the automated process, and it must be ensured manually, for this Burp Suite is used along with Turbo Intruder to test the servers for vulnerabilities and exploit any weaknesses found.

Figure 14 showcases a crucial part of the implementation phase, involving the use of Burp Suite for manual testing. The image depicts a sequence of requests, starting with a legitimate POST request followed by a smuggled GET request.



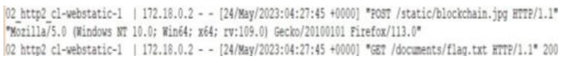
```
POST /static/blockchain.jpg HTTP/2
Host: localhost:8443
User-Agent: Mozilla/5.0 (Windows NT 10.0; Wi
Gecko/20100101 Firefox/112.0
Accept: image/avif,image/webp,*/*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Dnt: 1
Referer: https://localhost:8443/
Sec-Fetch-Dest: image
Sec-Fetch-Mode: no-cors
Sec-Fetch-Site: same-origin
Te: trailers
Content-Type: application/x-www-form-urlencc
Content-Length: 0

GET /documents/flag.txt HTTP/1.1
Host: localhost:8443
```

Fig. 14. Smuggled request for manual testing using Burpsuite

However, the subsequent GET request, positioned at the end of the request sequence, is a smuggled request. Its purpose is to retrieve the file "flag.txt" from the "/documents" directory. This request follows HTTP/1.1 protocol and lacks the required headers. Smuggled requests like this one exploit potential vulnerabilities in request handling, potentially bypassing security controls or accessing restricted files.

In Figure 15, the Docker log displays the response of both a normal request and a smuggled request. The log entry includes the IP address of the client (172.18.0.4), the timestamp of the request, and the details of each request.



```
02_http2_cl-webstatic-1  | 172.18.0.2 - - [24/May/2023:04:27:45 +0000] "POST /static/blockchain.jpg HTTP/1.1"
"Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0) Gecko/20100101 Firefox/113.0"
02 http2 cl-webstatic-1  | 172.18.0.2 - - [24/May/2023:04:27:45 +0000] "GET /documents/flag.txt HTTP/1.1" 200
```

Fig. 15. Docker Log Displaying Response of Normal Request and Smuggled Request

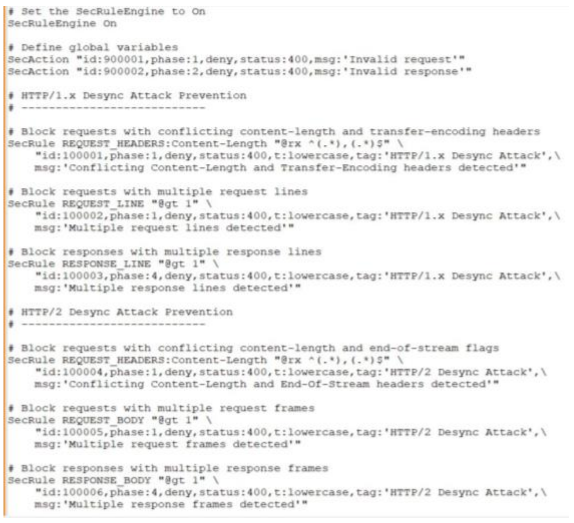
The first entry corresponds to a normal POST request, where the client requested the "/static/blockchain.jpg" file. The server responded with a status code of 200, indicating a successful request. The response size was 5348 bytes. The request was made from the URL https://localhost:8443/.

The second entry represents a smuggled GET request targeting the "/documents/flag.txt" file. Similar to the previous request, the server responded with a status code of 200, indicating success. The response size was 294 bytes. However, since this is a smuggled request, the specific details of the request headers and user agent are not provided in the log.

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.

RESULT AND ANALYSIS

Therefore, by exploiting HTTP/2 Content-Length vulnerability, we are able to access the forbidden file (flag.txt), and the tool that we used automated well in finding the vulnerability and reporting it. We have detected the HTTP Desync attack and also exploited the vulnerability in controlled environment which is critical in real world scenario, and also, we have written WAF rules for preventing certain requests from being sent to the server.



```
# Set the SecRuleEngine to On
SecRuleEngine On

# Define global variables
SecAction "id:900001,phase:1,deny,status:400,msg:'Invalid request'"
SecAction "id:900002,phase:2,deny,status:400,msg:'Invalid response'"

# HTTP/1.x Desync Attack Prevention
# ---------------------------

# Block requests with conflicting content-length and transfer-encoding headers
SecRule REQUEST_HEADERS:Content-Length "@rx ^(.*),(.*)$" \
    "id:100001,phase:1,deny,status:400,t:lowercase,tag:'HTTP/1.x Desync Attack',\
    msg:'Conflicting Content-Length and Transfer-Encoding headers detected'"

# Block requests with multiple request lines
SecRule REQUEST_LINE "@gt 1" \
    "id:100002,phase:1,deny,status:400,t:lowercase,tag:'HTTP/1.x Desync Attack',\
    msg:'Multiple request lines detected'"

# Block responses with multiple response lines
SecRule RESPONSE_LINE "@gt 1" \
    "id:100003,phase:4,deny,status:400,t:lowercase,tag:'HTTP/1.x Desync Attack',\
    msg:'Multiple response lines detected'"

# HTTP/2 Desync Attack Prevention
# ---------------------------

# Block requests with conflicting content-length and end-of-stream flags
SecRule REQUEST_HEADERS:Content-Length "@rx ^(.*),(.*)$" \
    "id:100004,phase:1,deny,status:400,t:lowercase,tag:'HTTP/2 Desync Attack',\
    msg:'Conflicting Content-Length and End-Of-Stream headers detected'"

# Block requests with multiple request frames
SecRule REQUEST_BODY "@gt 1" \
    "id:100005,phase:1,deny,status:400,t:lowercase,tag:'HTTP/2 Desync Attack',\
    msg:'Multiple request frames detected'"

# Block responses with multiple response frames
SecRule RESPONSE_BODY "@gt 1" \
    "id:100006,phase:4,deny,status:400,t:lowercase,tag:'HTTP/2 Desync Attack',\
    msg:'Multiple response frames detected'"
```

Fig. 16. Mod Security WAF Rules

In the context of Figure 16, Mod Security WAF Rules are depicted. These rules encompass a set of guidelines specifically designed for protecting web applications utilizing the HTTP/1.x and HTTP/2 protocols. The provided figure showcases a comprehensive collection of 29 distinct rules, which serve as an effective defense mechanism against potential web application attacks.

Overall, the implementation successfully simulated real-world web application scenarios and tested the effectiveness of the HTTP desync attack testing framework. By using Burpsuite 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.

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 17. 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%

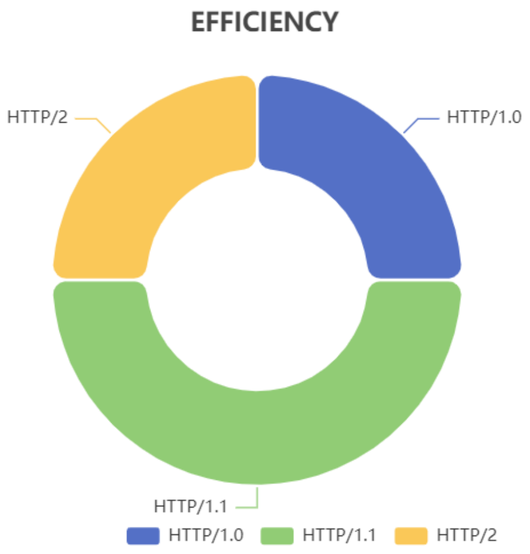


Fig. 17. Efficiency Comparison of HTTP Protocols: HTTP 1.0, 1.1, and 2.0

• 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.

BREAKDOWN OF HTTP PROTOCOL BASED ON USAGE

Figure 18 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.

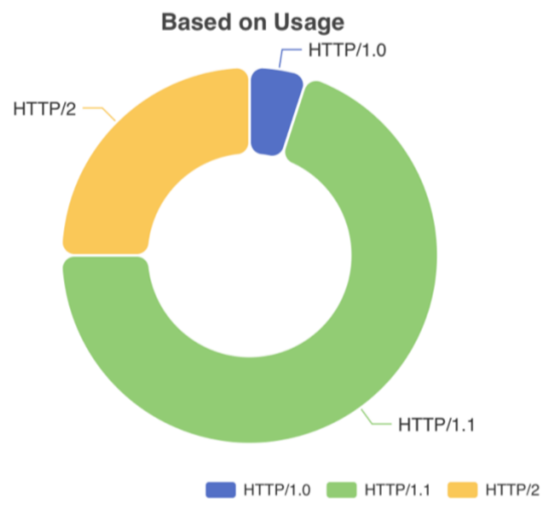


Fig. 18. Comparison of HTTP Protocols based on usage

• 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.

COMPARATIVE ANALYSIS OF TOP WAFs FOR PREVENTING HTTP DESYNC ATTACKS

The Figure 19 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.

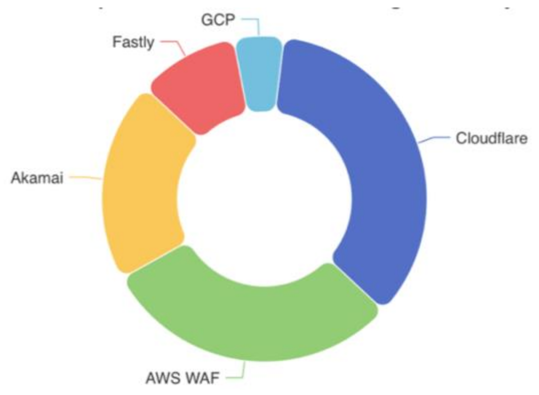


Fig. 19. Top WAFs used in the Industry

• 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.

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.

FUTURE WORK RECOMMENDATION

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] M. Y. Chiu, Y. F. Chen, and H. M. Sun, "Attacking Websites: Detecting and Preventing HTTP Request Smuggling Attacks," in Proceedings of the IEEE International Conference on Networking Architecture and Storage (NAS), Hindawi, pp. 1-6, 2022.

[2] C. Linhart, “HTTP-Request-Smuggling”, in Proceedings of the IEEE International Conference on Cybersecurity and Privacy (ICCP), pp. 123 -130, 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 Conference on Computer and Communications Security (CCS’21), pp. 1805 -1820, 2021.

[4] J. Kettle, "HTTP Desync Attacks: Smashing into the Cell Next Door," presented at the Black Hat USA, IEEE Access, vol. 9, pp. 57529 - 57531, 2019.

[5] R. Fielding and J. Reschke, Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content” presented at the IESG Conference, pp. 6 - 90, 2014.

[6] A. Klein, HTTP Request Smuggling In 2020 – New Variants, New Defenses and New Challenges”, presented at the Black Hat USA Conference, pp. 2-10, 2020.

[7] M. Grenfeldt, A. Olofsson, V. Engstrom, and R. Lagerstrom, “Attacking websites using HTTP request smuggling: empirical testing of servers and proxies,” in Proceedings of the 2021 IEEE 25th International Enterprise Distributed Object Computing Conference (EDOC), pp. 173 -181, IEEE, Gold Coast, Australia, October 2021.

[8] R. Leroy., Hiding Wookies in HTTP”, presented at the DEFCON 24. Conference, pp. 5-30, Aug 2016.

[9] C. Kern, "Securing the tangled web," in Communications of the ACM, vol. 57, no. 9, pp. 38-47, 2014.

[10] G. Tyson, R. Sommerville, L. N. B. Oliveira, and M. Mauthe, "Exploring HTTP header manipulation in-the-wild," in Proceedings of the 26th International Conference on World Wide Web, pp. 451- 458, 2017.

[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, 2015.

[12] R. Leroy, Checking HTTP Smuggling issues” in Proceedings of the Black Hat USA Conference, pp. 5-14, 2015.

[13] M. Doyhenard, Response Smuggling: Exploiting HTTP/1.1 Connections” presented at the DEFCON 29, pp. 5-20, 2020.

[14] R. Fielding and J. Reschke, Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing”, in the proceedings of the IESG Conference, pp. 5-71, 2014.

[15] J. Postel., Transmission Control Protocol”, IEEE Transactions on Communications, vol. 22, no. 5, pp. 637-648,1981.

[16] S. A. Mirheidari, A. Razaghpanah, H. Sardi, H. Haddadi, and P. Liatsis, "Web Cache Deception Escalates, " in Proceedings of the 31st USENIX Security Symposium, pp. 179-195, 2022.

[17] H. Nguyen, L. Iacono, H. Federrath et al. Your Cache Has Fallen: Cache-Poisoned Denial-of-Service Attack” at 26th ACM Conference on Computer and Communications Security (CCS), pp. 1915–1936, UK, 2019