Hindawi
Security and Communication Networks
Volume 2022, Article ID 3121177, 14 pages
https://doi.org/10.1155/2022/3121177

Research Article

Attacking Websites: Detecting and Preventing HTTP Request Smuggling Attacks

Qi-Xian Huang,[1] **Min-Yi Chiu**,[1] **Ying-Feng Chen**,[2] **and Hung-Min Sun**[3]

[1]*Institute of Information Systems and Applications, National Tsing Hua University, Hsinchu, Taiwan*
[2]*Institute of Information Security, National Tsing Hua University, Hsinchu, Taiwan*
[3]*Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan*

Correspondence should be addressed to Hung-Min Sun; hmsun@cs.nthu.edu.tw

Received 6 September 2022; Accepted 13 October 2022; Published 27 October 2022

Academic Editor: Kuo-Hui Yeh

Copyright © 2022 Qi-Xian Huang et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Until the development of HTTP request smuggling in 2005, individual HTTP requests were considered as independent entities and could not be split or merged. This is a security problem caused by inconsistent content length interpretation approach between web servers, or the web server is not fully implemented in accordance with the RFC standard. It is especially dangerous for web services with complex web architectures. It can route the victims to receive malicious responses, amplify the impact of certain low-threat vulnerabilities, steal user credentials, or bypass network devices' defenses. However, since its concept and implementation are quite difficult to overcome, it is often ignored by many network administrators, making users who browse such websites vulnerable to the HTTP request smuggling attacks. This paper proposes a general solution to deal with various HTTP request smuggling attacks. A reverse proxy implemented by Flask validates and cleans dubious HTTP requests from the client side and ensures that the original requests comply with RFC standards. Therefore, the website administrators no longer need to configure complicated network settings or customize some open-source project codes to resist or minimize the risk of the HTTP request smuggling attacks. A series of experiments demonstrate that this method is effective and practical.

1. Introduction

HTTP request smuggling is a unique attack first discussed in 2005 [1]. The attack forges an extra HTTP request inside a normal HTTP request, which is due to the differences in HTTP request interpretation between a front-end server and a back-end server. The problem occurs when one or more HTTP devices are running as a complex architecture such as web servers, cache servers, and proxy servers. In addition, HTTP request smuggling can be combined with other attack vectors and make a low-impact vulnerability request dangerous or make a non-vulnerability request vulnerable. For HTTP request smuggling attacks, an attacker must send specially made HTTP requests that cause different servers to diverge in parsing the requests, and the malicious requests can be passed to the target server without the knowledge of the other server.

In the groundbreaking research of 2005, three attack vectors of HTTP request smuggling were proposed. The first is the web cache poisoning, in which attackers can launch HTTP request smuggling attacks to manipulate the entries in a cache server and force regular users to access malicious contents while using the "poisoned" caches. The second is to bypass a firewall, which, due to the HTTP request smuggling characteristics, prevents firewall's security rules from being applied in smuggled requests because of its inability to parse the requests. This allows the attacker to introduce malicious content without being detected by a firewall.

The third is to assume that a user's TCP connection can be reused. An attacker can send a request to the web server by resorting to the next users' credentials connecting to the same web server or stealing other users' credentials by using XSS (Cross-Site Scripting), making reflected XSS more threatening.

1.1. Motivation. Research continued for almost a decade with little progress until 2015, when an article "Hiding Wookiees in HTTP" again attracted security researchers' interest. In 2019, a new attack technique of HTTP request smuggling was proposed on Black Hat 2019. This technique is to change the structure of Transfer-Encoding in HTTP protocol to achieve confusion. Then, in 2020, an article based on Content-Length variants again made outstanding contributions. In recent years, increasing research on HTTP request smuggling has indicated the need for a simple defense method.

1.2. Contributions. We here propose a method based on Flask as a reverse proxy to detect and prevent HTTP request smuggling. This method will require any user's request to comply with the RFC standard and pass the regex before the request is sent to the back-end server to examine whether the request is normal. This allows website operators to defend against such attacks without complicated installation, operation, or settings. This defense method is also easy to extend to other attack methods.

1.3. Organization. This paper is organized as follows. Section 2 presents the background for this paper, including HTTP request smuggling, reverse proxy, and RFC standard. Section 3 describes HTTP request smuggling related work in detail and proposes the advantages and disadvantages of various solutions. Section 4 presents the concept, method, and structure of the implementation and how it can effectively mitigate an HTTP request smuggling attack. In Section 5, evaluation results of our experiments are proposed and validated. Conclusions are provided in Section 6. Finally, future work is provided in Section 7.

2. Background

This section introduces the background, attack methods, threats, and some case studies of HTTP request smuggling in more detail. It also introduces the technologies and frameworks used in our defense system and the standards referenced by this defense system, including Flask, reverse proxy, regex, and RFC standard.

2.1. HTTP Request. In general, the normal process when user is browsing a site is as follows:

(1) The user sends a request to the front-end server (which can also be a load balancer or reverse proxy).

(2) The server forwards the request to one or more back-end servers.

(3) The website makes a response to the user.

A simple example is shown in Figure 1. This type of web architecture is very common and cannot be avoided in many situations.

In an HTTP request, these formats will be included: Request Line: Start with Method, then Request-URI and HTTP Protocol version, and finally end with CRLF.

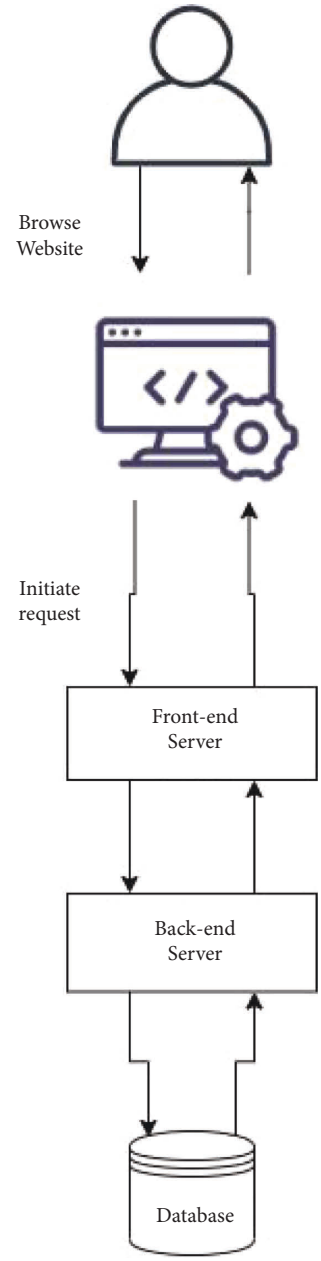

Figure 1: Example of HTTP request/response from front-end to back-end.

Headers: the header allows the client to pass the message about the request and the message of the client itself to the server. Basically, headers can be divided into four types:

(1) General header fields.

(2) Request header fields.

(3) Response header fields.

(4) Entity header fields.

After the end of Header feild, there will be an Empty Line. After the empty line, there will be the end of the Request or the beginning of the Message Body. Message body: this is not necessary in a request, but when

transmitting specific content or data (such as user account and password), it is usually used with the POST method.

The method in the Request Line will indicate the method to be executed for the Request-URI, and Figure 2 will list the request methods. Non-standard header field can also be customized by the developer. Figure 3 shows a request sent to https://google.com, illustrating the above content. Figure 4 is the response from https://google.com.

2.2. HTTP Request Smuggling. HTTP request smuggling is an attack vector that interferes with website processing procedures, where the website receives a series of HTTP requests from malicious users.

HTTP request smuggling is usually very dangerous because this vulnerability allows attackers to bypass security controls and access sensitive data without authorization and directly harm other users who use the application normally. Figure 5 shows the process of HTTP request smuggling and how this attack affects other users.

The vulnerability of HTTP request smuggling [2–6] itself is based on network communication of the website and has nothing to do with the program language used. Therefore, the website cannot defend HTTP request retrial system from the choice of program language. Because of the increasing size and complexity of the website structure, it is difficult for general developers to cover all possibilities. Thus, defense solution needs to be as simple as possible so the defense system does not become a weakness that attackers can exploit.

2.2.1. Impact. This paragraph discusses the harm and impact of some HTTP request smuggling attacks.

(1) Bypassing security control, some front-end servers are used as the first defense. If a request is allowed to pass, it will be forwarded to the back-end server, and the back-end server will not conduct further inspections. However, if the web server contains HTTP request smuggling vulnerabilities, this can be used to bypass or invalidate the front-end security system and smuggle malicious requests to the back-end server.

(2) Expose front-end request rewriting: some front-end servers will rewrite user requests before forwarding them to the back-end, such as adding some required headers. Attackers can use the functions provided by the web application itself to attack, such as adding another POST request to some data parameters, so the back-end server will process the smuggled request and respond to it, exposing sensitive information.

(3) Hijack other users' requests: if the web application stores and searches data, HTTP request smuggling can hijack or obtain the content of other users' requests, which may contain sensitive information, such as cookies or other user-sensitive information.

2.2.2. Type of HTTP Request Smuggling. To design a better defense system, understanding the types of HTTP request smuggling and attack methods is important, and the following section introduces common methods of exploiting. Figure 6 shows a simple CL-TE example, showing that both CL and TE are used in lines 7 and 8, and the value of CL is 14. This value (14) covers lines 11 through 13. However, since TE are also present, the value is Chunked and shows as 0 in line 11, which means the termination of TE. Therefore, data after line 12 will be regarded as the beginning of the next request. Figure 7 shows an example of TE-CL. Similarly, both CL and TE are used in lines 7 and 8, but here the value of CL is 3 which covers only line 11, and the rest is regarded as the beginning of the next request because TE also exists. This value is chunked and is specified as 9 in line 11, so the TE will regard the data to be transmitted this time as the content of line 12 to line 14.

CL-CL (also known as Double Content-Length attack) is relatively straightforward. It is the use of most web servers and middleware for the loose identification of the request body in the GET method request, which leads to the occurrence of vulnerabilities. Figure 8 shows that line 7 and line 8 are both CL but the values are different. If the proxy processes the first CL value first and includes line 11 as part of the body content, then when the back-end server is processing, the first CL is ignored and the second CL is processed first, leading to an HTTP request smuggling problem. Figure 9 shows an example of TE-TE. Since writing TE will confuse the web server, the web server can use parse CL instead of parse TE. Line 8 and line 9 show that there is a normal TE and a TE with a non-standard value. If the front-end server judges it as a wrong header and parses the CL, then it will become a CL-TE attack. Conversely, if the back-end server judges the TE as an incorrect header and parses the CL, it will become a TE-CL attack.

2.2.3. Case Study. To better understand the harm caused by HTTP request smuggling, this section presents an actual case.

Researcher Custodio 2020 discovered a large-scale account takeover that used CL-TE type HTTP request smuggling to steal session cookies. Force the victim to be hijacked and route into an open redirect.

(1) Use CL-TE vector to Poisoned Socket on slackb.

(2) Hijack the victim's request and change the request on slackb.com to use GET <url> HTTP/1.1.

(3) After the back-end server receives GET <url> HTTP/1.1, it will cause a 301 redirect <url> and carry the slack cookie.

(4) Use another server as the <url> in the exploit to steal the victim's cookie.

2.3. Flask. Flask is a lightweight web application microframework that uses Python language and is based on the Werkzeug WSGI (Web Server Gateway Interface)

Method	Description
GET	Transfer a current representation of the target resource.
HEAD	Same as GET, but only transfer the status line and header section.
POST	Perform resource-specific processing on the request payload.
PUT	Replace all current representations of the target resource with the request payload.
DELETE	Remove all current representations of the target resource.
CONNECT	Establish a tunnel to the server identified by the target resource.
OPTIONS	Describe the communication options for the target resource.
TRACE	Perform a message loop-back test along the path to the target resource.

Figure 2: Request methods.

```
GET / HTTP/1.1\r\n
Host: www.google.com\r\n
Connection: close\r\n
Sec-Fetch-Site: cross-site\r\n
Sec-Fetch-Mode: no-cors\r\n
Sec-Fetch-Dest: empty\r\n
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/85.0.4183.121
Safari/537.36\r\n
Accept-Encoding: gzip, deflate\r\n
Accept-Language: zh-TW,zh;q=0.9,en-US;q=0.8,en;q=0.7\r\n
\r\n
```

Figure 3: Sending a request to https://google.com.

```
HTTP/1.1 200 OK
Date: Tue, 15 Jun 2021 05:36:15 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=UTF-8
Strict-Transport-Security: max-age=31536000
BFCache-Opt-In: unload
P3P: CP="This is not a P3P policy! See g.co/p3phelp for more in
Server: gws
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN
Set-Cookie: 1P_JAR=2021-06-15-05; expires=Thu, 15-Jul-2021 05:3
Set-Cookie: NID=217=pM9dD7qlLhgMCV2EpYsJrt5VByvp-8gl0goIf1fAzZH
Alt-Svc: h3=":443"; ma=2592000,h3-29=":443"; ma=2592000,h3-T051
Connection: close
Content-Length: 115052

<!doctype html><html itemscope="" itemtype="http://schema.org/W
  <head>
    <meta charset="UTF-8">
    <meta content="origin" name="referrer">
    <meta content="/images/branding/googleg/1x/googleg_standard
    <link href="/manifest?pwa=webhp" crossorigin="use-credentia
    <title>
      Google
    </title>
    <script nonce="KC9Zo7PYSy6pB9EmnIDCnA==">
      (function(){
        window.google={
          kEI:'TzzIYK3cLZaFr7wP_7ClyAQ',kEXPI:'31',kBL:'bWXf'
        };
        google.sn='webhp';
        google.kHL='zh-TW';
      }
      )();
```

Figure 4: Response from https://google.com.

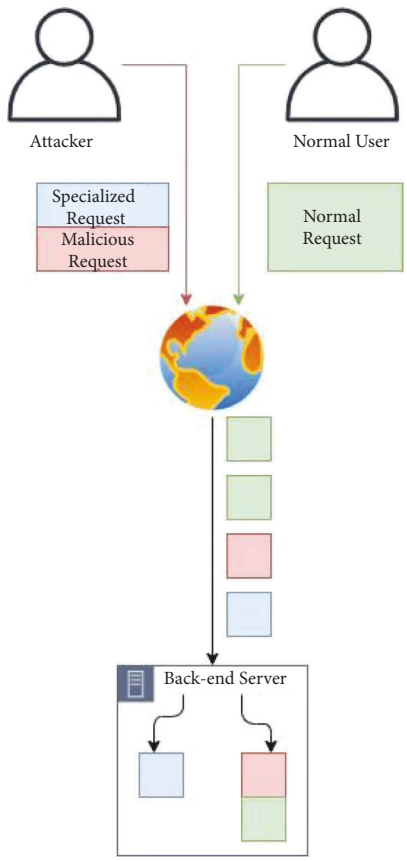

Figure 5: Flow of HTTP request smuggling.

application toolbox. Because it uses a simple core while maintaining scalability, we use it as the reverse proxy.

2.3.1. Abort. In the defense system, the abort function in Flask is used extensively since it can respond to interrupt the user's request, such as "404 Not Found." This defense plan refers to the RFC standard to make a "Bad Request" response to a problematic request with a status code of 400.

2.4. Reverse Proxy. In reverse proxy, the server will obtain resources from the associated web server based on the user's request, and then it returns them to the user. Some common functions of reverse are shown below, and Figure 10 shows a simple reverse proxy example.

(1) Hide the IP address of the rear web server from the user.

(2) As an application firewall, provide security requirements for the website.

(3) Load balancing, through URL redirection to the web server with lower resource load to obtain resources, and balance the pressure of all servers.

(4) Provide a cache service, so that when users want to obtain static content, they can get a faster response, and also reduce the load on the web server.

(5) Provide NAT traversal to the intranet.

2.5. Regular Expression. Often abbreviated as regex, this uses a single string to describe a string that meets a certain rule, that is, an expression that describes a certain rule. Figure 11 shows an example of regex. Since a large part of this defense system relies on determining the user's request, it is necessary to establish multiple sets of rules and make appropriate decisions based on 189 used scenarios, RFC standards, and some automated smuggling tools. Since this rule is the core of the entire defense system, it can neither allow legitimate users to misjudge when making special requests, nor allow attackers to bypass. Therefore, we have spent a lot of thought on the establishment of these rules.

2.6. RFC Standard. Request for Comment (RFC) is a series of memos issued by the IETF, mainly collecting information about the Internet. An RFC is issued after the review and is issued after a given number. The standard for the same subject is to replace the old RFC file with a new RFC file through a statement so as to update it, and the old RFC file will not disappear.

For example, the RFC 7230 standard declares that 2616 and 2145 are eliminated and 2817 and 2818 are updated. The content mainly describes the structure of the HTTP system and the interpretation of terms, while also defining the HTTP and HTTPS URI schemes, HTTP/1.1 message syntax, format and parsing methods, and security issues.

3. Related Works

Proposals concerning the detection and defense for HTTP request smuggling have been presented in the past years. For example, Watchfire [1] proposed using a strict HTTP parsing procedure for web servers for not reusing the same TCP connections after each request. Portswigger [7] made three suggestions: first, use HTTP/2 as the communication between front-end server and back-end server; second, enforce that the entire web architecture uses the same setting; and third, configure the front-end server to standardize the format of all problematic requests for detection. Portswigger has also developed a Burp Suite extension named HTTP request smuggler for checking HTTP request smuggling. This is an active checker that issues a series of requests to targets that cause a vulnerable system to timeout in order to verify the attack is successful. Another detection technology is Ontology, which presents a method of utilizing semantic technology in web application security, and Munir et al. [8] propose the HTTP protocol ontology to mitigate the communication protocol-related attacks. Klein [9] proposed a detection and defense technology at the TCP level, which is based on another proxy server or web server. SafeBreach Labs [10] adopted a two-layer architecture. The first layer is Socket Abstraction Layer (SAL), which implements multiple hooks on the socket functions and collects any incoming network bytes. The second layer is the HTTP/1.x request smuggling firewall (RSFW). This layer will force each request to comply with the HTTP/1.x standard to fulfill the requirement of protection for HTTP request smuggling.

```
POST / HTTP/1.1 \r \n
Host: exmaple.com \r \n
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/85.0.4183.121
Safari/537.36 \r \n
Accept-Encoding: gzip, deflate \r \n
Accept-Language: zh-TW,zh;q=0.9,en-US;q=0.8,en;q=0.7 \r \n
Content-Type: application/x-www-form-urlencoded \r \n
Content-Length: 14 \r \n
Transfer-Encoding: chunked \r \n
Connection: keep-alive \r \n
 \r \n
0 \r \n
 \r \n
smuggling \r \n
```

Figure 6: Example of CL-TE.

```
POST / HTTP/1.1 \r \n
Host: exmaple.com \r \n
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/85.0.4183.121
Safari/537.36 \r \n
Accept-Encoding: gzip, deflate \r \n
Accept-Language: zh-TW,zh;q=0.9,en-US;q=0.8,en;q=0.7 \r \n
Content-Type: application/x-www-form-urlencoded \r \n
Content-Length: 3 \r \n
Transfer-Encoding: chunked \r \n
Connection: keep-alive \r \n
 \r \n
9 \r \n
 \r \n
smuggling \r \n
 \r \n
0 \r \n
```

Figure 7: Example of TE-CL.

```
GET / HTTP/1.1 \r \n
Host: exmaple.com \r \n
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/85.0.4183.121
Safari/537.36 \r \n
Accept-Encoding: gzip, deflate \r \n
Accept-Language: zh-TW,zh;q=0.9,en-US;q=0.8,en;q=0.7 \r \n
Content-Type: application/x-www-form-urlencoded \r \n
Content-Length: 9 \r \n
Content-Length: 0 \r \n
Connection: keep-alive \r \n
 \r \n
smuggling \r \n
 \r \n
```

Figure 8: Example of CL-CL.

The design principle hinges on whenever each request containing malicious content is found, the socket will be immediately closed and not forwarded to the web server [11–18].

In other related research, an article [19] in 2015 once aroused people's interest in HTTP request smuggling. This article provided detailed introductions to the attack vectors and exploits of HTTP request smuggling, which are also pivotal to our experiment. Mantoro [20] proposed a mechanism to detect and defend web attacks through the reverse proxy. Although it is mainly aimed at SQL Injection, its concept still brings us some inspiration.

4. Methodology

4.1. Core Components. Since the method we propose must strictly meet requirements of the HTTP protocol and request header in the RFC standard, we briefly describe the concept and structure of the HTTP request below.

```
1 POST / HTTP/1.1 \r \n
2 Host: exmaple.com \r \n
3 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
  AppleWebKit/537.36 (KHTML, like Gecko) Chrome/85.0.4183.121
  Safari/537.36 \r \n
4 Accept-Encoding: gzip, deflate \r \n
5 Accept-Language: zh-TW,zh;q=0.9,en-US;q=0.8,en;q=0.7 \r \n
6 Content-Type: application/x-www-form-urlencoded \r \n
7 Content-Length: 3 \r \n
8 Transfer-Encoding: chunked \r \n
9 Transfer-Encoding: cow \r \n
10 Connection: keep-alive \r \n
11  \r \n
12 D \r \n
13  \r \n
14 smuggling \r \n
15  \r \n
16 0 \r \n
17
```

Figure 9: Example of TE-TE.

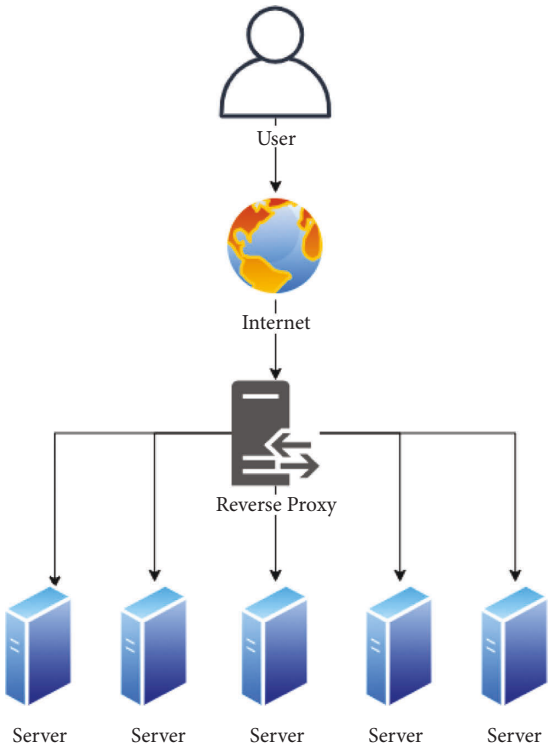

Figure 10: Example of reverse proxy.

4.1.1. HTTP Request. It must include the start line, protocol, header, and body of the client's message to the server.

4.1.2. Start-Line. Start with a method (such as POST, GET, and OPTIONS), then request URI and protocol version (such as HTTP/1.1), and end with CRLF (newline).

4.1.3. Header. Define HTTP operating parameters. Non-standard HTTP headers defined as needed are allowed. However, some headers are necessary, and a lack of them may cause failure. The header can be divided into general header, request header, and entity header. This article mainly discusses Transfer-Encoding in the general header and Content-Length in the entity header.

4.1.4. Request Body. This carries entities related to the response; however, depending on the request method, the request body may not be needed. The request header related to HTTP request smuggling is discussed below.

4.1.5. Connection. According to RFC7230 [21] (Section 6.1), Keep-Alive is used by default in HTTP/1.1, which allows multiple requests to be carried on the same connection. It appears in the header field "Connection: Keep-Alive" of the request, which tells the server not to close the TCP connection after receiving the request. Thus, only one TCP handshake is needed, and continuing to use the same TCP connection reduces loading on the server. A keep-alive connection is shown in Figure 12.

4.1.6. Transfer-Encoding (TE). Transfer-Encoding indicates what type of conversion is applied to the message body. There five main attributes: chunked, compress, deflate, gzip, and identify. Here we discuss chunked, which wraps the payload body and is used to divide it into a series of chunks for transmission.

4.1.7. Content-Length (CL). Unlike Transfer-Encoding, Content-Length is an entity header, representing the length of the request body sent to the receiver in decimal. HTTP request smuggling attack is based on inconsistent parsing and processing between front-end server and back-end server (or any middleware such as a reverse proxy) for the length of HTTP request. These differences enable attackers to insert one HTTP request into others and achieve smuggling.

A simple example is shown in Figure 13, a POST request, including CL and TE. When a website with this vulnerability receives a request and the front-end server does not support

Figure 11: Example of regex.

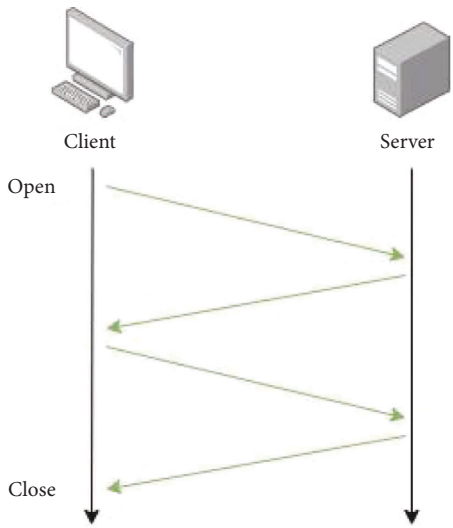

Figure 12: Keep-alive connection.

```
POST /test HTTP/1.1 \r \n
Host: smugging-example.com \r \n
Content-Length: 6 \r \n
Transfer-Encoding: chunked \r \n
\r \n
0 \r \n
\r \n
G
```

Figure 13: Example of HTTP request smuggling.

chunked, it resolves the request according to the CL, which sends the entire request to the back-end server. When the back-end server receives it, it supports TE and parses the request body as chunked. When faced with zero, it deems that the request has terminated and uses the unused G as the beginning of the subsequent request. It is also necessary to know all variants of requests that may lead to HTTP request smuggling, so the requirements and processing specifications of RFC for exceptional cases of the request are discussed below.

From RFC 7230 [21], specification when TE and CL exist at the same time, TE has a higher priority than CL, so CL should be removed before the Request is forward. A 400 error should be returned if any non-standard value exists in the TE header. For RFC 2616 [22], it specifies that a 400 error will be returned whenever there are two TE or CL. However, in RFC 7230, explicitly states that only a valid CL field is allowed to be reserved.

4.2. System Architecture. Figure 14 shows the architecture of our system, and Figure 15 shows the system flowchart of our proposed scheme. The system consists of the following components.

(i) *Router*. This module receives and intercepts all requests from the client side. Intercepted requests are passed to the Checker. If the Controller finds no exception, it uses the function as a reverse proxy to pass the request to the web server.

(ii) *Controller*. When this module receives an intercepted request, it calls the Checker to determine if the request is an exception. If the Controller gets a "False" from Checker, it returns a 400 Bad Request or keep one valid header while removing redundant headers.

(iii) *Checker*. This module defines whether TE and CL headers are exceptions according to the RFC standard. In addition to the header value, the headers themselves need to be checked. For this, the regex is used to check the header itself, and the header value is simply the format defined by the RFC standard. If the value does not belong to the five values of TE, it is considered as an exception. The Checker also checks if there is more than one TE or CL mixed in a request and logs such requests as exceptions. Though CL or TE is not necessary for some attack vectors, it can still cause HTTP request smuggling, such as request splitting using NULL character injection. Therefore, the Checker will also detect whether the header and request body is abnormal. If any of the inspections fail, the Checker will return False to the Controller.

(iv) *Executor*. After the Controller instructs this module, the specified header is deleted from the underlying WSGI environment of Flask to ensure that no mixed or duplicated headers will be transmitted to the web server.

5. Experimental Result

In this section, we manually send the problematic request to evaluate whether or not the defense system is as expected.

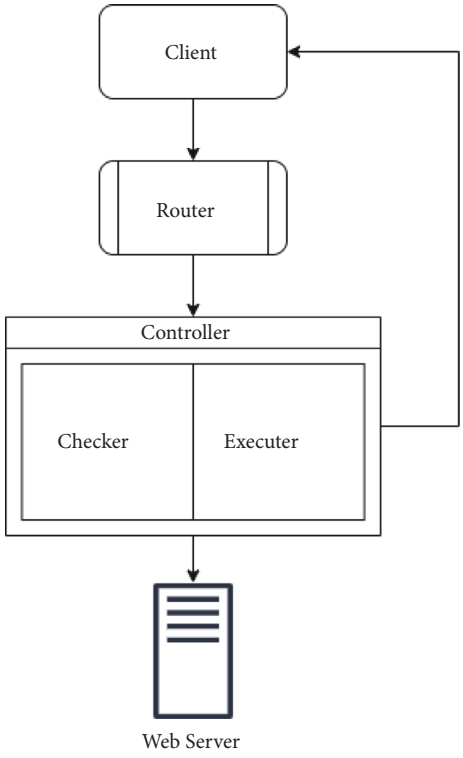

Figure 14: System architecture.

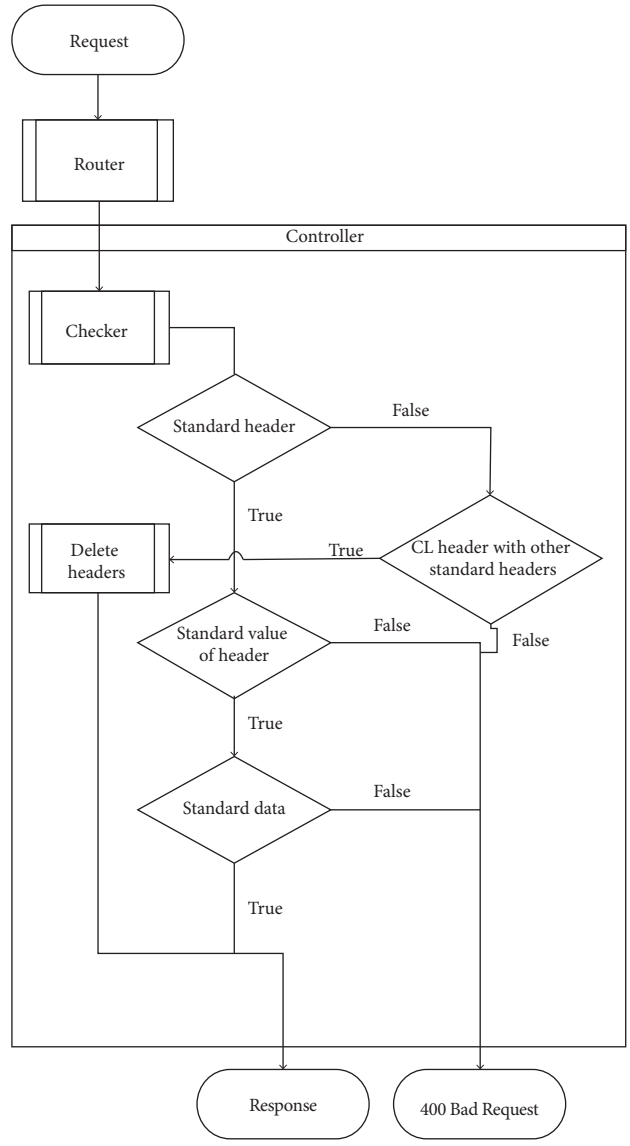

Figure 15: System flowchart.

Figure 16 shows that the checker detected a non-standard TE value, Response 400 error, and marked it as a WARNING to log the request. In the TE header field, the value xchunked is used to mislead some web servers. Assuming that the front-end treats xchunked as chunked and legal but the back-end treats it as invalid, this difference in interpretation may cause HTTP request smuggling. Also, Figure 17 shows the content of the problematic TE header in our system log. The difference between Figures 17 and 16 is that the TE header itself, instead of the value, is the problem. Figure 18 shows the original request in the green line area, which sends both TE and CL headers to a server. The red line area indicates that the CL is detected and needs to be removed. The last orange section shows that CL has been removed from the underlying WSGI environment of Flask, leaving only Transfer-Encoding.

Next, the experimental settings and payloads for testing are selected from Yu [23], which was originally designed to investigate the exploitation of HTTP request smuggling attacks, but we took advantage of these environments to evaluate our solution and test it with different HTTP request smuggling attacks. The five experiment scenarios are listed below.

(i) Lab1: HaProxy1.6, Apache Traffic Server 6.2.2/7.1.1, and the latest version of Nginx.

(ii) Lab2: Apache Traffic Server 7.1.2, mattrayner/lamp-1084 and fbraz3/lnmp 7.1.

(iii) Lab3: HaProxy 2.0 and Gunicorn 20.0.4.

(iv) Lab4: Nginx1.17.6 (CVE-2019-20372).

(v) Lab5: Jetty 9.4.9 (CVE-2017-7656).

More information can be obtained from [23]. The defense system uses Flask as the reverse proxy with a port in 5000. Experimental results are shown below.

Figure 19 shows that the request is directly routed to ATS/7.1.1 (port 8007) when the defense system is not activated. The section outlined in green is the payload for testing Lab1. Although there are no CL or TE headers, this is still one variant of the HTTP request smuggling attack vectors, and the first query contains two bad headers. Usually, this payload is regarded as two queries, but it is clear that ATS interprets it as having three queries and sends back three responses: a 400 Invalid HTTP request, a 200 OK, and 404 Not Found. Figure 20 shows the situation after executing our defense system that prevents the GET method from containing any request body with data. One example is highlighted under "Request Input Stream." The

```
WARNING:root:
Host: 127.0.0.1:5000
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:84.0) Gecko/20100101 Firefox/
84.0
Content-Type: application/x-www-form-urlencoded
Transfer-Encoding: xchunked
Connection: keep-alvie
Upgrade-Insecure-Requests: 1


INFO:werkzeug:127.0.0.1 - - [05/Feb/2021 16:38:46] "[31m[1mPOST /test HTTP/1.1[0m" 400 -
```

Figure 16: Non-standard value of TE header detected in our system log.

```
WARNING:root:
Host: 127.0.0.1:5000
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:84.0) Gecko/20100101 Firefox/84.0
Content-Type: application/x-www-form-urlencoded
Transfer-Encodingx: chunked
Connection: keep-alvie
Upgrade-Insecure-Requests: 1


INFO:werkzeug:127.0.0.1 - - [04/Feb/2021 17:53:43] "[31m[1mPOST /test HTTP/1.1[0m" 400 -
```

Figure 17: Non-standard value of TE header detected in our system log.

```
WARNING:root:Origin request:
Host: 127.0.0.1:5000
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:84.0) Gecko/20100101 Firefox/84.0
Content-Type: application/x-www-form-urlencoded
Transfer-Encoding: chunked
Content-Length: 7
Connection: keep-alvie
Upgrade-Insecure-Requests: 1

WARNING:root:delete header: CONTENT_LENGTH

WARNING:root:deleted:
{'wsgi.version': (1, 0), 'wsgi.url_scheme': 'http', 'wsgi.input':
<werkzeug.serving.DechunkedInput object at 0x7fbab0f21ee0>, 'wsgi.errors': <_io.TextIOWrapper
name='<stderr>' mode='w' encoding='utf-8'>, 'wsgi.multithread': True, 'wsgi.multiprocess':
False, 'wsgi.run_once': False, 'werkzeug.server.shutdown': <function
WSGIRequestHandler.make_environ.<locals>.shutdown_server at 0x7fbab0f2b040>, 'SERVER_SOFTWARE':
'Werkzeug/1.0.1', 'REQUEST_METHOD': 'POST', 'SCRIPT_NAME': '', 'PATH_INFO': '/test',
'QUERY_STRING': '', 'REQUEST_URI': '/test', 'RAW_URI': '/test', 'REMOTE_ADDR': '127.0.0.1',
'REMOTE_PORT': 58767, 'SERVER_NAME': '127.0.0.1', 'SERVER_PORT': '5000', 'SERVER_PROTOCOL':
'HTTP/1.1', 'HTTP_HOST': '127.0.0.1:5000', 'HTTP_USER_AGENT': 'Mozilla/5.0 (Macintosh; Intel Mac
OS X 10.15; rv:84.0) Gecko/20100101 Firefox/84.0', 'CONTENT_TYPE': 'application/x-www-form-
urlencoded', 'HTTP_TRANSFER_ENCODING': 'chunked', 'HTTP_CONNECTION': 'keep-alvie',
'HTTP_UPGRADE_INSECURE_REQUESTS': '1', 'wsgi.input_terminated': True, 'werkzeug.request':
<Request 'http://127.0.0.1:5000/test' [POST]>}
INFO:werkzeug:127.0.0.1 - - [04/Feb/2021 18:20:39] "[37mPOST /test HTTP/1.1[0m" 200 -
```

Figure 18: Mixed-use case of CL header and TE header.

```
printf 'GET /something.html?zorg=1 HTTP/1.1\r\n'\
'Host: dummy-host7.example.com\r\n'\
'X-Something: "\0something"\r\n'\
'GET http://dummy-host7.example.com/index.html?replacing=1&zorg=2 HTTP/1.1\r\n'\
'\r\n'\
'GET /targeted.html?replaced=maybe&zorg=3 HTTP/1.1\r\n'\
'Host: dummy-host7.example.com\r\n'\
'\r\n'\
| nc 127.0.0.1 8007
HTTP/1.1 400 Invalid HTTP Request
Date: Mon, 08 Feb 2021 07:54:16 GMT
Connection: keep-alive
Server: ATS/7.1.1


HTTP/1.1 200 OK
Server: ATS/7.1.1


HTTP/1.1 404 Not Found
Server: ATS/7.1.1
```

Figure 19: Payload and response of Lab1 without our system.

```
WARNING:root:
[!]Method GET with request body
Request Input Stream:
GET /targeted.html?replaced=maybe&zorg=3 HTTP/1.1
Host: dummy-host7.example.com


INFO:werkzeug:127.0.0.1 - - [08/Feb/2021 15:43:13] "[31m[1mGET /something.html?zorg=1
HTTP/1.1[0m" 400 -
```

Figure 20: Log content of Lab1 generated from our system.

```
printf 'GET / HTTP/1.1\r\n'\
'Host: lnmp.com\r\n'\
'aa: \0GET /2333 HTTP/1.1\r\n'\
'Host: lnmp.com\r\n'\
'\r\n'\
| nc 127.0.0.1 9010 |grep -E "HTTP|Server"
HTTP/1.1 400 Invalid HTTP Request
Server: ATS/7.1.1
HTTP/1.0 400 Invalid HTTP Request
Server: ATS/7.1.1
```

Figure 21: Payload and response of Lab2 without our system.

```
WARNING:root:Request header contains request start line:
Host: lnmp.com,lnmp.com
Aa: GET /2333 HTTP/1.1

INFO:werkzeug:127.0.0.1 - - [09/Feb/2021 16:26:56] "[31m[1mGET / HTTP/1.1[0m" 400 -
```

Figure 22: Log content of Lab2 generated from our system.

```
printf 'POST / HTTP/1.1\r\n'\
'Host: localhost\r\n'\
'Transfer-Encoding:\x0b chunked\r\n'\
'Connection:keep-alive\r\n'\
'Content-Length: 65\r\n'\
'\r\n'\
'0\r\n'\
'\r\n'\
'POST / HTTP/1.1\r\n'\
'Host: localhost\r\n'\
'Content-Length: 1\r\n'\
'\r\n'\
'0'\
| nc 127.0.0.1 9014
HTTP/1.1 200 OK
Server: gunicorn/20.0.4
Date: Mon, 08 Feb 2021 12:43:49 GMT
Connection: keep-alive
Content-Type: text/html; charset=utf-8
Content-Length: 0

HTTP/1.1 200 OK
Server: gunicorn/20.0.4
Date: Mon, 08 Feb 2021 12:43:49 GMT
Connection: keep-alive
Content-Type: text/html; charset=utf-8
Content-Length: 1
```

Figure 23: Payload and response of Lab3 without our system.

```
WARNING:root:
[!]The value of Transfer-Encoding is not allow
Host: localhost
Transfer-Encoding:
 chunked
Connection: keep-alive
Content-Length: 65

INFO:werkzeug:127.0.0.1 - - [09/Feb/2021 12:20:57] "[31m[1mPOST / HTTP/1.1[0m" 400 -
```

Figure 24: Log content of Lab4 generated from our system.

```
printf 'GET /a HTTP/1.1\r\n'\
'Host: localhost\r\n'\
'Content-Length: 56\r\n'\
'\r\n'\
'GET /_hidden/index.html HTTP/1.1\r\n'\
'Host: notlocalhost\r\n'\
'\r\n'\
|nc 127.0.0.1 9015
HTTP/1.1 302 Moved Temporarily
Server: nginx/1.17.6
Date: Mon, 08 Feb 2021 04:45:36 GMT
Content-Type: text/html
Content-Length: 145
Connection: keep-alive
Location: http://example.org

<html>
<head><title>302 Found</title></head>
<body>
<center><h1>302 Found</h1></center>
<hr><center>nginx/1.17.6</center>
</body>
</html>
HTTP/1.1 200 OK
Server: nginx/1.17.6
Date: Mon, 08 Feb 2021 04:45:36 GMT
Content-Type: text/html
Content-Length: 22
Connection: keep-alive

This should be hidden!%
```

Figure 25: Payload and response of Lab4 without our system.

```
WARNING:root:
[!]Method GET with length header
GET /a? HTTP/1.1
Host: localhost
Content-Length: 56
```

Figure 26: Log content of Lab4 generated from our system.

area outlined in green in Figure 21 is the payload used to request HTTP smuggling tests on Lab2. This attack vector is called "Request Splitting by NULL Character Injection,"

```
printf 'POST /?test=4973 HTTP/1.1\r\n'\
'Transfer-Encoding: chunked\r\n'\
'Content-Type: application/x-www-form-urlencoded\r\n'\
'Host: localhost\r\n'\
'\r\n'\
'100000000\r\n'\
'\r\n'\
'POST /?test=4974 HTTP/1.1\r\n'\
'Content-Length: 5\r\n'\
'Host: localhost\r\n'\
'\r\n'\
'\r\n'\
'0\r\n'\
'\r\n'\
|nc 127.0.0.1 9014|grep -E "HTTP/1.1|Server"
HTTP/1.1 200 OK
Server: Jetty(9.4.9.v20180320)
HTTP/1.1 200 OK
Server: Jetty(9.4.9.v20180320)
```

Figure 27: Payload and response of Lab5 without defense system.

which is a technique that uses NULL character to make ATS end queries prematurely, and ATS will not close the connection that continues the second query. The ATS server responds 400 Invalid HTTP Request twice. The first time it responded for aa: \0, and the second time the request did not comply with the HTTP request standard. It can be seen that in Figure 22, the system effectively prevents this situation.

The green line area in Figure 23 is the payload for testing Lab3, which is a case of mixed use of TE and CL and causes the Gunicorn server to return 200 OK twice. It should be noted that "Transfer-Encoding: \textbackslash x0b chunked" will be treated by Haproxy as a request. Therefore, Haproxy ignores TE header and uses CL header to parse the body, not dropping TE header but passing the request to the back-end server. Figure 24 shows that our defense system determines this request as a non-standard TE value and returns 400 Bad Request.

The green outlined area of Figure 25 is the payload used to test Lab4, and the response is shown below. The server returned a 302 Moved Temporarily and a 200 OK. The result of using our defense system in Lab4 is presented in Figure 26. The system here can detect any GET method containing CL headers and return 400 Bad Request.

The green outlined line area in Figure 27 is the payload used to test Lab5. The attack vector is called Chunk size attribute truncation. Jetty will treat 100000000 as 0, so there will be two responses. Figure 28 shows that the expected 400 Bad Request return is actually a 500

```
INFO:werkzeug:127.0.0.1 - - [08/Feb/2021 18:21:19] "[35m[1mPOST /test=4973 HTTP/1.1[0m"
500 -
```

Figure 28: The response of Lab5 with our system.

INTERNAL SERVER ERROR, which may be because the Flask itself cannot be too long or have a non-standard TE length. However, this smuggling payload will not be forwarded to the back-end server, so the defense was still successful.

6. Conclusion

In the increasingly complex web architecture, attacks against HTTP Protocol are more and more diverse. In this study, we introduce a novel method to defend against HTTP request smuggling. This approach provides an efficient scheme that can be applied to almost any web server and makes web servers more secure without complex configuration. Although this methodology is just a proof of concept, it shows potential to be applied to mitigate other similar attacks as well.

7. Recommendations for Future Work

This research mainly presents a solution for HTTP request smuggling, which can be effective in real-world scenarios. However, because we do not use large-scale and complex websites, as in real-world scenarios, if the architecture is complex or a website has high traffic, there may be performance issues requiring improvement. In addition, this system can be further modified to incorporate load balancing, so it has the potential for future development.

Data Availability

The data used to support the findings of this study can be accessed from the following website: https://github.com/ZeddYu/HTTP-Smuggling-Lab.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was supported in part by the Ministry of Science and Technology, Taiwan, under project nos. MOST110-2221-E-007-040-MY3 and MOST111-2221-E-007-078-MY3.

References

[1] R. Heled, *HTTP-Request-Smuggling*, Portswigger, Chelford Road, Knutsford, 2005.

[2] 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.

[3] K. James, *Http Desync Attacks: Request Smuggling Reborn. PortSwigger Web Security Blog*Portswigger, Chelford Road, Knutsford, 2019.

[4] K. James, *Password Theft login.newrelic.Com via Request Smuggling*, HackerOne, San Francisco, California, 2019.

[5] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "Nezha: efficient domain-independent differential testing," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP)*, Jose, CA, USA, May 2017.

[6] J. Chen, J. Jiang, H. Duan, N. Weaver, T. Wan, and V. Paxson, "Host of troubles: multiple host ambiguities in HTTP implementations," in *Proceedings of the ACM Conference on Computer and Communications Security*, Vienna, Austria, October 2016.

[7] J. Kettle, *HTTP Desync Attacks: Smashing into the Cell Next Door*, Portswigger, Black Hat USA, 2019.

[8] R. F. Munir, N. Ahmed, A. Razzaq, A. Hur, and F. Ahmad, "Detect HTTP specification attacks using ontology," in *Proceedings of the 2011 Frontiers of Information Technology*, pp. 75–78, IEEE, Islamabad, Pakistan, December 2011.

[9] A. Klein, "Technical note: detecting and preventing http response splitting and http request smuggling attacks at the tcp level," 2005, https://www.securityfocus.com/archive/1/408135.

[10] A. Klein, *HTTP Request Smuggling In 2020 – New Variants, New Defenses and New Challenges*, Portswigger, Black Hat USA, 2020.

[11] E. E. Han, "Detection of web application attacks with request length module and regex pattern analysis," in *Proceedings of the International Conference on Genetic and Evolutionary Computing*, pp. 157–165, Springer, NewYork, NY, USA, August 2015.

[12] M. Grenfeldt, A. Olofsson, V. Engström, and R. Lagerström, "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.

[13] J. Chen, J. Jiang, H. Duan, N. Weaver, T. Wan, and V. Paxson, "Host of troubles: multiple host ambiguities in http implementations," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, Vienna, Austria, October 2016.

[14] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel, "SWAP: mitigating XSS attacks using a reverse proxy," in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, pp. 33–39, IEEE, Vancouver, BC, Canada, May 2009.

[15] C. H. Lin, J. C. Liu, and C. C. Lien, "Detection method based on reverse proxy against web flooding attacks,"vol. 3, pp. 281–284, in *Proceedings of the 2008 8th International Conference on Intelligent Systems Design and Applications*, vol. 3, pp. 281–284, IEEE, Kaohsuing, Taiwan, November 2008.

[16] A. Lamba, "Analysing sanitization technique of reverse proxy framework for enhancing database-security," *International Journal of Information and Computing Science*, vol. 1, no. 1, 2014.

[17] A. Razzaq, Z. Anwar, H. F. Ahmad, K. Latif, and F. Munir, "Ontology for attack detection: an intelligent approach to web application security," *Computers & Security*, vol. 45, pp. 124–146, 2014.

[18] 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.

[19] 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/.

[20] T. Mantoro, "Log visualization of intrusion and prevention reverse proxy server against Web attacks," in *Proceedings of the 2013 International Conference on Informatics and Creative Multimedia*, pp. 325–329, IEEE, Kuala Lumpur, Malaysia, September 2013.

[21] R. Fielding and J. Reschke, *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*, RFC-editor, California, 2014.

[22] R. Fielding, J. Gettys, J. Mogul et al., *Hypertext Transfer Protocol – HTTP/1.1*, RFC-editor, California. , https://www.rfc-editor.org/info/rfc2616, 1999.

[23] Z. Yu, "HTTP-Smuggling-Lab - github," 2019, https://github.com/ZeddYu/HTTP-Smuggling-Lab.

[24] R. Fielding and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content," 2014, https://www.rfc-editor.org/rfc/rfc7231.txt.

[25] E. Custodio, "smuggler-GitHub," 2021, https://github.com/defparam/smuggler.