Less is Often More: Header Whitelisting as Semantic Gap Mitigation in HTTP-Based Software Systems

Andre Büttner, Hoai Viet Nguyen, Nils Gruschka, Luigi Lo Iacono

Less is Often More: Header Whitelisting as Semantic Gap Mitigation in HTTP-based Software Systems

Andre Büttner[1][0000−0002−0138−366X], Hoai Viet Nguyen[2][0000−0002−6540−5389], Nils Gruschka[1][0000−0001−7360−8314], and Luigi Lo Iacono[2][0000−0002−7863−0622]

[1] University of Oslo, Gaustadalléen 23B, 0373 Oslo, Norway
{andrbut,nilsgrus}@ifi.uio.no
[2] Hochschule Bonn-Rhein-Sieg, Grantham-Allee 20, 53757 Sankt Augustin, Germany
luigi.lo_iacono@h-brs.de

Abstract. The web is the most wide-spread digital system in the world and is used for many crucial applications. This makes web application security extremely important and, although there are already many security measures, new vulnerabilities are constantly being discovered. One reason for some of the recent discoveries lies in the presence of intermediate systems—e.g. caches, message routers, and load balancers—on the way between a client and a web application server. The implementations of such intermediaries may interpret HTTP messages differently, which leads to a semantically different understanding of the same message. This so-called semantic gap can cause weaknesses in the entire HTTP message processing chain.
In this paper we introduce the header whitelisting (HWL) approach to address the semantic gap in HTTP message processing pipelines. The basic idea is to normalize and reduce an HTTP request header to the minimum required fields using a whitelist before processing it in an intermediary or on the server, and then restore the original request for the next hop. Our results show that HWL can avoid misinterpretations of HTTP messages in the different components and thus prevent many attacks rooted in a semantic gap including request smuggling, cache poisoning, and authentication bypass.

Keywords: HTTP · web · intermediaries · semantic gap · security · header whitelisting

1 Introduction

When the web was created more than 30 years ago, no one had imagined that it would evolve into a global system used by billions of people for nearly every aspect of their daily lives. Due to the ever-growing number of users, web servers need to be offloaded to meet performance and scalability requirements. This is often realized by intermediate systems such as caches, which store static resources, or load balancers, which distribute requests across different server instances [10].

However, these various HTTP-based entities involved in the message processing pipeline can also induce problems. Web applications often suffer from differences in the processing of HTTP messages. It can lead to serious security vulnerabilities if the processing elements in the processing chain interpret the same message differently [31, 4]. In this context, the HTTP header fields take on an important role, as they are essential for interpreting an HTTP message. Unfortunately, in practice they are not handled consistently by HTTP implementations, which can lead to, e.g., different perceptions of the syntactic validity or caching behavior of an HTTP message. Moreover, an HTTP header can include standardized and non-standardized header fields; both are permitted by the HTTP standard [12]. However, non-standardized header fields are ignored by some components, while having a decisive role for others, especially when it comes to access privileges. If the components are not complementing each other well, unintentional behavior can occur, which can consequently be abused by a malicious user.

The underlying problem of different processing and interpretation of HTTP messages by different processing units within a processing pipeline is called "semantic gap" [31, 4]. Although this issue has been known for a long time, new types of semantic gap attacks are continuously being discovered [2]. Also, current security mechanisms like WAFs can only partly mitigate this threat (see Section 4). Therefore, attackers nowadays have a good chance to exploit a semantic gap for malicious purposes.

This paper presents a novel protection means that specifically addresses the semantic gap problem. Our analysis of known semantic gap vulnerabilities shows that most of them result from inconsistent processing of HTTP request header fields. We therefore suggest normalizing and filtering request header fields before they are processed by HTTP components. By passing only those header fields that are required by a particular intermediary and that can be reliably processed, attacks involving broken, malformed, or non-standardized header fields can be prevented. This can be used to defend against not only known attacks but also potential zero-day exploits, as is already being done for malware [35, 28]. We provide the following main contributions:

1. The semantic gap in HTTP message processing is defined, known attacks that exploit the semantic gap are analyzed, and they are categorized according to their causes.
2. The concept of header whitelisting (HWL) is introduced as a measure to mitigate the semantic gap. A prototype implementation is presented and evaluated, showing effective protection against known attacks.

The remainder of this paper is structured as follows: Section 2 details the semantic gap in the context of HTTP-based software systems. Further foundations in terms of real-world attacks are presented in Section 3 followed by Section 4, which reviews the related work of proposed measures against these attacks. In Section 5 the HWL approach is introduced and its prototype implementation is described. In Section 6, we experimentally evaluate the effectiveness of HWL

with respect to semantic gap attack evasion. The paper closes with a discussion in Section 7 and a conclusion with an outlook on future work in Section 8.

2 Semantic Gaps in HTTP Message Processing

Even though the HTTP protocol is specified in RFC standards, HTTP-based software systems tend to suffer from semantic differences when processing HTTP messages, which is summarized by the term *semantic gap*. Within the scope of this work, the semantic gap is defined as *inconsistent processing of HTTP messages inside a pipeline between the actual application logic and the intermediaries*. Such a behaviour can have serious consequences, as will be discussed in more detail in Section 3.

We identified three main causes of inconsistent HTTP message processing. The first one is ambiguous wording within the HTTP standard. It is generally forbidden, e.g., to include more than one header field with identical field name in HTTP messages. However, the HTTP standard leaves room for *well-known exceptions*. This ambiguity leads to widely varying HTTP implementations. Some may reject a request with duplicate header fields. Others will accept such requests and consider either the first or the last one and either ignore or remove the other instance. Furthermore, no limit is defined for the length of the header fields [12]. Both aspects can lead to a wide range of vulnerabilities, such as Request Smuggling or HTTP Header Oversize.

Another cause of inconsistent HTTP message processing is improper HTTP implementations. This is especially relevant for parsing the HTTP header. There are implementations that allow invalid syntax and ignore the affected header fields. Others clean up requests from invalid header fields, which in turn can affect subsequent HTTP-processing components. And yet other implementations completely reject requests with invalid syntax. If an intermediary and a server handle invalid meta characters in HTTP header fields differently, this can be exploited, e.g., to cause a denial-of-service.

A final major cause for a semantic gap is different HTTP versions used by the components involved or different specifications for the same version. For the widely used HTTP/1.1, e.g., there exists the outdated RFC 2616 [11] and the current RFC 7230 [12]. This results in implementations that conform to the outdated version, while others conform to the latest standard. Developers and server providers are certainly aware of this fact. Nonetheless, it is possible that there is still software in use that refers to the deprecated specification. This can be critical since RFC 2616 does, e.g., not explicitly forbid trailing whitespaces in header field names, while RFC 7230 requires the HTTP message to be rejected in this case. Accordingly, this can lead to HTTP Desync attacks. Another example is the line folding option that allows to span a header field value over multiple lines. This is supported in RFC 2616, but is deprecated in RFC 7230. It was demonstrated that this can be exploited for Request Smuggling. Additionally, there are also discrepancies between different HTTP versions. It has been shown, e.g., that a client can cause various types of denial-of-service

attacks in cases where an intermediary supports HTTP/2 while the web server uses HTTP/1.1 [15]. This is due to header compression in HTTP/2, which is not supported in HTTP/1.1. In this case, a client sends header fields to the intermediary via HTTP/2 compression. Since these header fields are transmitted to the web server via HTTP/1.1, they must be decompressed and can be significantly larger. Therefore, there is an increased server load, which can lead to other connections being blocked.

In summary, there are many causes of semantic gaps in an HTTP message processing pipeline. These cannot be solved right away or easily, as it would require harmonizing all HTTP implementations to one single unambiguous specification. This is unrealistic considering how many HTTP-based software components are available and in use in the wild. As we will emphasize in the subsequent Section 3, effective means are nonetheless urgently required for web application developers and providers to cope with the security threats and risks stemming from semantic gaps rooted vulnerabilities.

3 Attacks Rooted in a Semantic Gap

In recent years, the semantic gap has been the root for many serious threats in web-based layered software systems that consist of various intermediaries. In this section, we provide an overview of semantic gap vulnerabilities in HTTP message processing as defined in Section 2. Attacks based on semantic gaps in other application layers, such as processing multiple cookies [3], are out of scope.

The *Response Splitting* attack [23] was one of the first vulnerabilities to exploit a semantic gap to perform web cache poisoning. Here, an attacker takes advantage of a parsing issue that occurs when carriage return (CR) and line feed (LF) characters are not sanitized or escaped properly. If a web server then reflects a value of the request in the response, an attacker can exploit both issues by sending a request with CR and LF characters in conjunction with a malicious response hidden in a header field value. The reflected malicious input forces the returned response to be interpreted as two responses. The second response, which is completely under the attacker's control, is then stored by the cache, effectively poisoning it with the attacker's malicious payload.

The *Request Smuggling* attack [27] exploits a semantic gap in parsing more than one `Content-Length` header fields—although forbidden according to RFC 7230 [12]—to smuggle a hidden request through an intermediary. With this technique, a malicious client can provoke a web cache poisoning if two intermediaries (one of these a cache) pick different `Content-Length` header fields and therefore read different amounts of the payload. The ambiguous interpretation of `Content-Length` header fields can also be applied to hide malicious requests from security intermediaries such as WAFs, Intrusion Detection Systems (IDS) or access control mechanisms. In a rather new variant of Request Smuggling, called *HTTP Desync* attack [22] similar effects can be achieved using the `Transfer-Encoding` or non-standardized headers like `X-Forwarded-Host`.

The *Host of Trouble* (HoT) attack [4] is another attack that aims to poison web caches or bypass security policies, e.g. in a WAF. Unlike Request Smuggling, this attack uses duplicate `Host` header fields. Although the presence of more than one `Host` header field is not compliant with RFC 7230, Chen et al. [4] uncovered many real-world HTTP implementations that ignore this requirement.

The *Cache-Poisoned Denial of Service* (CPDoS) attack [31] exploits semantic gaps to deny access to web resources by poisoning the cache with error pages. The three CPDoS attack variants *HTTP Header Oversize* (HHO), *HTTP Meta Character* (HMC), and *HTTP Method Override* (HMO) were introduced that exploit the mismatch between header size limits, meta character handling, and the method override header respectively. The authors showed in empirical studies that millions of web sites were vulnerable to CPDoS. Nathan Davison presented another variant of CPDoS by using the Hop-by-Hop header mechanism [7].

The *Web Cache Deception* (WCD) attack [14] aims to disclose sensitive information with the help of a cache. This can be achieved in cases where caches decide whether to store responses based on the URL and consider URLs with suffixes such as `.css` or `.png` as static. The attacker appends such suffixes to URLs of resources containing confidential information, which is then stored in the cache. In a 2020 analysis of the Alexa Top 5K, Mirheidari et al. found 340 web sites vulnerable to WCD attacks [30].

In addition to the attacks mentioned above, new attacks that exploit a semantic gap are published very frequently, e.g. [2]. Also, we found that some of the reported vulnerabilities can be exploited in different ways. For example, the `X-Original-URL` and `X-Rewrite-URL` header fields can be used for CPDoS attacks or the Hop-by-Hop mechanism can be used to cause Request Smuggling.

Table 1. List of attacks that exploit a semantic gap in the processing of HTTP messages inside a pipeline between the actual application logic and intermediate systems

Attack		**Semantic Gap**	**Embedment**
Response Splitting		Meta character handling	URL
Request Smuggling		Content-Length, Transfer-Encoding, X-Forwarded-Host, X-Host, X-Forwarded-Server, X-Forwarded-Scheme, X-Original-URL, X-Rewrite-URL, Meta character handling	Header
Host-of-Trouble		Host header	Header
CPDoS	HHO	Header size limit	Header
	HMC	Meta character handling	Header
	HMO	Method overriding headers, e.g. X-HTTP-Method-Override, X-HTTP-Method, X-Method-Override	Header
	others	X-Original-URL, X-Rewrite-URL	Header
Hop-by-Hop		Connection header	Header
WCD		URL parsing	URL

This overview of attacks based on a semantic gap emphasizes their high relevance, especially when considering the flourishing number of attack variants. Their impact on real applications highlights the urgent need for efficient mitigation. As we already pointed out in Section 2, there is no easy way to eliminate the root cause of semantic gaps. However, as the summary view of Table 1 shows,

in many cases the processing of the HTTP header is the starting point for the attacks. This suggests that a broad range of semantic gap based attacks can be mitigated by treating the HTTP header in some suitable manner. Before elaborating this observation further, we will first review the literature on proposed countermeasures to identify possible approaches to mitigate such vulnerabilities.

4 Related Work

Most of the literature about the attacks discussed in Section 3 also suggests mitigation measures. WCD attacks may be prevented if HTTP responses contain proper caching directives. A further proposed measure is to put static files into a separate directory and configure the web server or cache to only allow caching of the contents of this directory. In general, invalid URL paths should be handled with an error response [14]. CPDoS attacks can be avoided by configuring a cache in a way that HTTP responses with certain error status codes are not stored. Additionally, the `no-store` directive in error responses by the server application could prevent the cache from storing them [31]. A more stricter HTTP parsing could help to mitigate Request Smuggling [27]. To this end, e.g., a proof-of-concept implementation exists that hooks into a server's socket functions, monitors HTTP messages, and closes the connection if HTTP violations are detected [24]. This approach in particular enforces valid formatting of header fields and adds special treatment to the `Content-Length` and `Transfer-Encoding` headers. While this helps to mitigate a broad range of Request Smuggling attacks and other attacks based on invalid meta characters, cache poisoning, Hop-by-Hop and Host-of-Trouble vulnerabilities are not prevented. Also, the usage of HTTP/2 mitigates Request Smuggling issues due to the use of binary frames and streams [22]. Hop-by-Hop vulnerabilities are prevented by this as well, since the `Connection` header field is not used in HTTP/2. However, this does not solve vulnerabilities based on non-standardized header fields or HoT. Response Splitting can be avoided by validating input from the client, especially in query parameters, and by removing special characters from a string before including it in a response header value [23]. To prevent `Host` header field attacks, Chen et al. recommend to ensure compliance with RFC 7230 [4]. According to them, the attack is the result of incorrect implementations. They refer to the latest HTTP/1.1 specification which should, in contrast to RFC 2616, define more clearly how to deal with ambiguities of the host. For mitigating access control vulnerabilities, access restrictions should be defined through the web application and for each resource separately [29].

Web application firewalls (WAFs) are intermediaries that intend to prevent many different types of attacks against web applications. They can be operated in different ways, such as a (transparent) reverse proxy [19], a network bridge [21], embedded in the web application [9] or as a cloud service [20]. Different measures can be included in a WAF, for example input validation, protocol enforcement, authorization or cookie signatures [8, 6, 16]. There are whitelist, blacklist or hybrid models that can be applied to define rules for HTTP traffic [5]. Several

WAFs provide default configurations that include mitigations against many attacks including the OWASP Top Ten [32]. This gives WAF users a basic level of protection and certainly prevents several types of attacks. Nevertheless, it does not provide absolute protection. Basically WAFs are also intermediaries that can be subject to semantic differences. There are reports about bypassing WAF rules, for instance by using Request Smuggling techniques [25] or non-standardized header fields [1]. Consequently, WAFs are also affected by the semantic gap. Another problem is the restrained use due to the complexity and required effort of configuring and updating WAF configurations [33].

In summary, the measures proposed in the literature are very fragmented and only apply to a specific type of attack. Hence, a comprehensive protection against semantic gap attacks is hard to achieve and requires to careful adopt all of the discussed countermeasures specifically tailored for each environment. As such, this is very error-prone and cannot always be implemented consistently for the complete processing chain. Mirheidari et al. take this line and suggest that there should be a different view of web application security [30]. They recommend not to focus on individual HTTP components but to have a holistic view of the entire system. We encourage this perspective as well and assume that the semantic gap is an important factor for the security of web applications regarding all HTTP components involved. We present such a more holistic approach to mitigating semantic gap attacks in the following section.

5 Header Whitelisting

As we noted in Section 3, almost all known semantic gap attacks have their roots in HTTP header parsing ambiguities (see Table 1). This suggests that many of the attacks can be thwarted by a strictly standard compliant message parsing. Since a comprehensive consistent implementation is a practically hopeless endeavor given the large number of different HTTP components that exist in practice, other approaches are required to counter the attacks. By introducing the so-called *header whitelisting* (HWL) we suggest that a specialized security intermediate normalizes the header using the HTTP standard as a baseline and reduces it to the minimum header fields required for processing by a particular component in the processing pipeline. In this paper we focus on attacks that are based on malicious request header fields. Therefore our approach is only applied to the request header.

Whitelisting is used for some time for various types of security mechanisms [34]. Network firewalls use whitelists to filter network traffic [17], for example. Spam filters use domain name server whitelists (DNSWL) that provide a list of trusted mail servers and IP addresses [26]. Header Whitelisting, as we introduce it in this paper, means that a request is transformed to consist only of required—and preferably standardized—header field names with expected header field values. When HWL is applied in front of an HTTP-processing entity, this entity will only receive HTTP requests whose headers are reduced to the fields it knows and minimally needs. As a result, requests containing malformed header entries

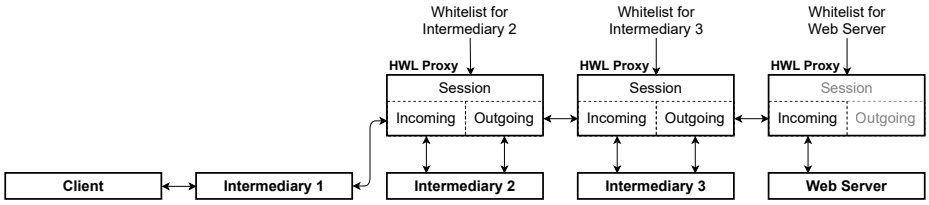

Fig. 1. Basic architecture of an HWL equipped HTTP-based software system. Intermediate systems and/or the web server are wrapped by HWL proxies tailoring the upstream HTTP message as individually required by each HWL-protected component.

are rejected or the affected entries are removed by HWL before they reach actual processing nodes. In addition, reducing the amount of header entries to the minimum necessary ensures that unknown or hidden functionality cannot be exploited. This is especially important if e.g. web frameworks are used in the application, as these contain a large number of functions and standard behaviors that are often neither known nor needed, but can potentially be used for attacks.

5.1 Architecture

Since every component that processes HTTP requests may process header fields differently, the idea is to isolate single components separately from invalid and needless header fields. This can be achieved by applying HWL to some or all of the components in the message processing pipeline, each with an individual whitelist (see Figure 1). An HWL Proxy is introduced that can be wrapped around intermediaries and/or the application logic running on the server, enforcing an individual whitelist for each wrapped component. It consists of three core modules. The *Incoming* module receives the requests and applies header whitelisting. This means, it checks whether the HTTP header fields match with the configured whitelist and consequently removes all header fields that are not listed. The resulting request containing only whitelisted header fields is then forwarded to the protected component. The *Outgoing* module receives back the processed request from the protected component and restores the original request respecting modifications that are possibly done by the component. This request is then forwarded to the next HTTP component in the chain. Note that in case the HWL Proxy is deployed in front of a web application server, the Outgoing module is not required, because the server is the last instance that receives an HTTP request and therefore does not forward it further. The *Session* module handles the linking between a request in the Incoming module to the corresponding request in the Outgoing module and temporarily stores the removed header fields.

Our approach requires to modify a request before it enters and before it leaves a HWL-protected component. A request received by the HWL Proxy will be normalized and customized according to the underlying whitelist. After the request has been processed by the component, the HWL Proxy restores the

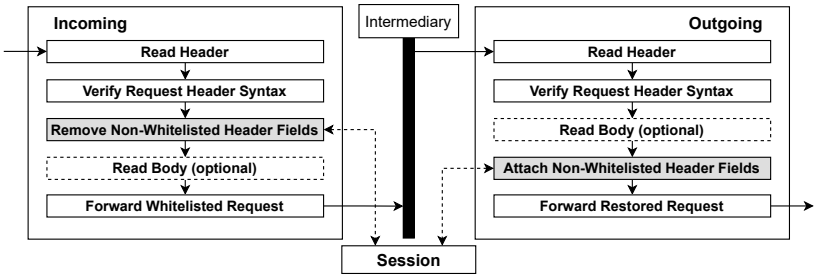

Fig. 2. HWL Proxy implementation showing the different processing steps for HTTP requests. Essential steps for the header whitelisting are highlighted in grey.

original request and sends it to the next hop on the path. This can not be realized with a common proxy architecture, as common proxies only process a request or response once. Hence, to implement a proof-of-concept common proxies including most of the WAFs could not be used as starting point. We therefore implemented the HWL Proxy as a separate application. By this, it can be deployed as an extension to existing intermediaries and server applications.

5.2 Implementation

The HWL Proxy is implemented in Go and is available as open source[3]. Note that its standard http library was not used because it is subject to vulnerabilities as well [2]. Figure 2 shows an overview of the implementation.

Whitelist Specification. The whitelist for the HWL Proxy is defined as an array of item objects, specified in a JSON format. In a real-world scenario, this whitelist would be created by the operators of the intermediaries or web server, who are conscious about which header fields are required. The whitelist items contain a string parameter `key` which represents the header field name. In addition, each whitelist item can contain an optional second string parameter `val`. This can be used to specify an allowed header field value or range of values with a regular expression. If this parameter is not specified, any value allowed by the HTTP specification is accepted. An example whitelist that only accepts `Host`, `Connection` and `Content-Length` header fields may be specified as follows:

```
[
    {"key": "host"},
    {"key": "connection", "val": "(close|keep-alive)"},
    {"key": "content-length", "val": "\\d+"}
]
```

[3] `https://github.com/digital-security-lab/hwl-proxy`

HTTP Request Processing. The header of a received HTTP request is read in and then parsed and verified according to the RFC 7230.Specifically, the syntax of the request line, the following header fields, and the end of the header section is validated. If a violation of the standard is found, the HWL Proxy sends a `400 Bad Request` error response back to the client and closes the connection. This might occur, e.g., if the header contains invalid meta characters or if the header section is not terminated properly with a blank line. Otherwise, the whitelist is applied to the request header. The Incoming module iterates through all header fields and appends the current one to the list of whitelisted header fields, if it matches one of the specified items contained in the whitelist. Otherwise, it is added to the list of non-whitelisted header fields. Note that whitelisted header fields that appear multiple times in the original request but did not cause an error during request validation are also included multiple times in the list of whitelisted header fields. As mentioned in Section 2, RFC 7230 generally prohibits the use of duplicate header fields, thus the HWL Proxy takes care of this requirement. However, since exceptions are allowed an operator can explicitly specify expected duplicates in the whitelist configuration.

The header fields that are not whitelisted are kept in a temporary session so that the Outgoing module can access them later to reconstruct the original request. After the header whitelisting has been applied, the body of the HTTP request is processed (if any). If the whitelisted header fields contain a valid `Transfer-Encoding: chunked` or `Content-Length` header field, the body is read in accordingly. In case both of these header fields are present, the `Content-Length` header field is removed to avoid HTTP Desync attacks. If none of these header fields is present, no body is expected. As for the header parsing, in case any violation is detected, the HWL Proxy sends a `400 Bad Request` error response directly to the client and closes the connection. Finally, the normalized and whitelisted HTTP request is delivered to the wrapped component.

The reception and verification of the HTTP header in the Outgoing module are identical to the according steps in the Incoming module. This applies also to the body. However, since the body parsing depends on the occurrence of certain header fields such as `Content-Length` and `Transfer-Encoding`, the body is received before attaching the non-whitelisted header fields to ensure the body always maintains the same size and encoding. After processing the body the non-whitelisted header field names are each compared with the header field names of the request received from the intermediary. A non-whitelisted header field is not appended to the request when the same header field was set by the intermediary. This is done to prevent duplicates that may occur in case the intermediary appended a header field that was already included in the original request received by the client.

The final step of this module is to forward the request containing the request line, the whitelisted header fields including the modifications made by the intermediary and the non-whitelisted header fields with the mentioned exceptions.

Table 2. List of the developed and analyzed test cases including the attacks and the software that were used within the test environment (`<SP>`: whitespace character).

ID	Attack Type	Causing Header	Intermediary	Web Server
TC1	Request Smuggling	`Content-Length`	ATS 7.1.2	NodeJS 4.1.2
TC2	Request Smuggling	`Transfer-Encoding` + `<SP>`	ATS 7.1.2	NodeJS 4.1.2
TC3	Request Smuggling	`X-Rewrite-Url`	NGINX 1.1.15	Symfony 3.4.0
TC4	CPDoS	`X-Original-Url`	Varnish 6.3.1	Symfony 3.4.0
TC5	CPDoS	`X-HTTP-Method-Override`	Varnish 6.3.1	Play 1.5.0
TC6	Hop-by-Hop	`Connection`	Varnish 3.0.0	NodeJS 4.1.2
TC7	HoT	`Host`	ATS 7.1.2	Rails 5.2.0

Table 3. Test results for all seven test cases with all possible test setups (○: HWL disabled, ●: HWL enabled, ⊕: attack prevented ⊖: attack succeeded)

Intermediary	Server	TC1	TC2	TC3	TC4	TC5	TC6	TC7
○	○	⊖	⊖	⊖	⊖	⊖	⊖	⊖
○	●	⊕	⊕	⊕	⊕	⊕	⊖	⊕
●	○	⊕	⊕	⊖	⊖	⊖	⊕	⊖
●	●	⊕	⊕	⊕	⊕	⊕	⊕	⊕

HTTP Response Processing. As discussed and argued in the beginning of this section, HWL is applied to HTTP requests only. Therefore, the Incoming and Outgoing modules forward the unmodified response to previous HTTP component in the processing chain.

6 Evaluation

To evaluate our HWL approach, we recreated the attacks presented in Section 3 caused by irregularities in header fields in a lab environment, deployed our solution, and evaluated the effect of header whitelisting. The test environment consists of a web server, one intermediary and a client. This was implemented using three virtual server instances running Ubuntu 16.04 LTS. Different proxies and server application software were installed and configured in accordance with the original attack description. In total we created seven different test cases representing different attack constellations (see Table 2). Attacks that are not covered here are discussed in Section 7.

For each test case, an attack vector was created that causes a malicious behavior. Furthermore, it was defined what response would be expected and what response represents an unintentional behavior. Intermediary and web server software were selected that are vulnerable to the respective attack type.

As shown in Table 3, the seven test cases were executed and analyzed in four different setups with and without header whitelisting deployed at the intermediate and server to illustrate the effect of HWL on the attack. Every test case is defined by a certain request or sequence of requests. A simple command line

application was developed as the client. It sends sequences of HTTP requests specified in text files and stores requests and received responses into result files for subsequent analysis. The requests sent were logged to ensure that the test was executed correctly, while the logged responses were used to distinguish between legitimate requests and successful or averted attacks.

Table 3 also shows the results of all combinations from the seven test cases and the four setups. The first setup (HWL Proxy deployed but not enabled) illustrates that all attacks were successfully replicated in our lab and that the HWL Proxy does not interfere with communication when disabled. The other three setups include at least one component—i.e., either the intermediary, the server or both—with header whitelisting enabled. The attacks in all seven test cases except from TC6 were prevented when HWL was applied at least on the web server. When HWL was applied on the intermediary only, the attacks in TC1, TC2 and TC6 were prevented. When HWL was applied to both intermediary and web server, all seven attacks were prevented successfully.

Through this experimental evaluation, we can show that the proposed HWL approach is an effective countermeasure against all seven attacks considered in the test cases. These tested attacks span all known attacks that can be traced back to irregularities in the header. Thus, HWL can be considered as the first approach that mitigates a broad variety of semantic gap attacks including Request Smuggling, Host-of-Trouble, and CPDoS. Although this is no guarantee that unknown attacks will also be prevented, the test results show that previously unknown request header attacks can be thwarted by HWL's request header normalization and minimization when the server and all intermediaries are wrapped by an HWL Proxy.

7 Discussion

As the evaluation shows, HWL is a promising new approach to closing most of the publicly known – and possibly some not yet known – semantic gaps in an HTTP message processing chain. These strengths, limitations, and other considerations are discussed below.

Strengths. The main objective of the proposed approach has been achieved. All attacks considered were successfully prevented by whitelisting and normalizing request header fields. Even if whitelisting was only applied to either the intermediary or the web server, some attacks could still be prevented. The best protection is achieved when HWL is applied to both the intermediaries and the server.

Another advantage of the HWL Proxy comes from its architecture, as it can be wrapped around arbitrary HTTP components. This eliminates ambiguities when parsing HTTP messages without having to change every single implementation.

HWL also potentially helps to prevent unknown attacks related to a semantic gap. The risk that new vulnerabilities based on semantic gaps can be successfully

exploited is mitigated by HWL by transforming the HTTP request header to a minimal and standards-compliant equivalent before processing by potentially vulnerable HTTP components.

Limitations. An obvious limitation of the proposed HWL is its restriction to attacks that target request header fields. The semantic gap, however, can occur in all parts of HTTP messages. Web Cache Deception, e.g., occurs when the query string of a URL is manipulated. Also, Response Splitting is rather caused by query parameters than by request header fields. However, most of the known semantic gaps relate to request header fields (see Table 1) and for these HWL provides a first coherent protection mechanism.

The evaluation presented in Section 6 omits those attacks that are not related to the HTTP header, as preventing them is not the scope of this paper (see Section 3). Furthermore, only those CPDoS attacks were included that are due to certain non-standardized header fields. The HMC CPDoS variant is covered by test case TC2, which includes an invalid space character to achieve Request Smuggling. The HHO CPDoS variant was excluded, because it cannot be prevented by the HWL Proxy implementation described in Section 5. This would require the consideration of the header size, which may be included to the whitelist specification in future work.

Today CDN services are commonly used intermediaries for improving the performance of web applications. However, they were not included in the test cases for evaluation. The main reason is that the attacks considered, such as Request Smuggling, could not be recreated with the available services, as they have already applied patches against such attack vectors. Nevertheless, vulnerable CDNs will behave similar to the caches in our test environment and therefore should benefit from the use of HWL likewise.

Vulnerabilities. The introduced HWL Proxy is a novel component that may contain vulnerabilities in itself. Parsing errors may occur, e.g., which provide an additional attack surface for semantic gaps. This can be avoided by considering language-theoretic security approaches, which aim to make input validation more secure and advise against *ad hoc* methods [13]. When this is reliably applied to the HWL Proxy, robust message parsing is propagated to the components that the HWL Proxy protects.

In addition, an implementation of the HWL Proxy may be vulnerable to denial-of-service (DoS) attacks, especially if requests with many non-whitelisted header fields must be processed frequently. Depending on the implementation, this can lead to a heavy load on the HWL Proxy. Our current HWL Proxy prototype implementation does not restrict header field processing and is therefore potentially vulnerable to DoS attacks.

Whitelist Specification. The whitelist is a determining factor in the effectiveness of HWL. An incorrect whitelist configuration can lead to false-positives

and thus to malfunction of the processing pipeline and the entire application. Furthermore, a cache may not work properly in case relevant header fields are not included in the whitelist. However, we assume that the risk here is lower compared to WAFs that are typically more complex [33]. A further measure to avoid false-positives can be to monitor the traffic in the testing phase when deploying the HWL Proxy in order to detect too restrictive policies.

There are concepts for WAFs to create rules autonomously during the test phase of applications [36, 33]. This could be transferred to the header whitelisting as well. The HWL Proxy may learn which header fields are actually used and automatically create an appropriate whitelist.

Similar to WAFs, a default configuration for the whitelist proxy should be provided. For instance, a default whitelist could be created that contains all header fields from the IANA Message Header registry [18]. These are standardized in RFC documents and should therefore not be critical in most cases while providing a broad compatibility.

Deployment. In future work, it could be considered to standardize the whitelist approach. It may even be added to existing HTTP libraries. The only requirement to ensure its effectiveness is that the header whitelisting is applied before any header is processed.

Another option is to provide the HWL Proxy as Software-as-a-Service in a cloud. As this is already common for WAFs operated by e.g. CloudFront, Cloudflare or Akamai, this can be realized similarly for the header whitelisting. This requires only an appropriate routing and the possibility for a customer to configure the whitelist.

8 Conclusion and Outlook

In this paper, we presented and categorized attacks on Web applications that exploit the semantic gap of HTTP interpretation. Based on the observation that many of these attacks are based on malicious HTTP request headers, we introduced the concept of Header Whitelisting. The idea of this approach is to filter all but a predefined set of HTTP headers before HTTP intermediaries and HTTP servers. The evaluation of the prototype implementation showed that all tested attacks could be prevented successfully.

In the future, it is conceivable to standardize such a mechanism and to include it to actual HTTP-based software systems. In addition, the performance of this approach should be thoroughly investigated to identify implementation strategies with the least performance impact. Finally, advanced features like automatic whitelist generation, access control lists or processing of response headers shall be considered.

References

1. Bijjou, K.: Web application firewall bypassing – how to defeat the blue team (2015), https://owasp.org/www-pdf-archive/OWASP_Stammtisch_Frankfurt_-

_Web_Application_Firewall_Bypassing_-_how_to_defeat_the_blue_team_-_2015.10.29.pdf
2. BitK: I found another way to do HTTP smuggling, https://twitter.com/BitK_/status/1351587043814604805
3. Calzavara, S., Rabitti, A., Bugliesi, M.: Sub-session hijacking on the web: Root causes and prevention. Journal of Computer Security **27**(2), 233–257 (2019)
4. Chen, J., Jiang, J., Duan, H., Weaver, N., Wan, T., Paxson, V.: Host of troubles: Multiple host ambiguities in http implementations. In: 23th ACM SIGSAC Conference on Computer and Communications Security (CCS) (2016)
5. Clincy, V., Shahriar, H.: Web application firewall: Network security models and configuration. In: IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC) (2018)
6. Consortium, W.A.S., et al.: Web application firewall evaluation criteria, version 1.0 (2006)
7. Davison, N.: Abusing http hop-by-hop request headers (2019), https://nathandavison.com/blog/abusing-http-hop-by-hop-request-headers
8. Dermann, M., Dziadzka, M., Hemkemeier, B., Hoffmann, A., Meisel, A., Rohr, M., Schreiber, T.: Best practices: Use of web application firewalls. The Open Web Application Security Project, Tech. Rep (2008)
9. Desmet, L., Piessens, F., Joosen, W., Verbaeten, P.: Bridging the gap between web application firewalls and web applications. In: 4th ACM workshop on Formal methods in security (2006)
10. Dikaiakos, M.D.: Intermediary infrastructures for the World Wide Web. Computer Networks **45**(4), 421–447 (Jul 2004)
11. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, IETF (1999), https://tools.ietf.org/html/rfc2616
12. Fielding, R., Reschke, J.: Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230, IETF (2014), https://tools.ietf.org/html/rfc7230
13. Ganty, P., Köpf, B., Valero, P.: A language-theoretic view on network protocols. In: International Symposium on Automated Technology for Verification and Analysis. pp. 363–379. Springer (2017)
14. Gil, O.: WEB CACHE DECEPTION ATTACK. In: Blackhat USA (2017), https://blogs.akamai.com/2017/03/on-web-cache-deception-attacks.html
15. Guo, R., Li, W., Liu, B., Hao, S., Zhang, J., Duan, H., Sheng, K., Chen, J., Liu, Y.: CDN Judo: Breaking the CDN DoS Protection with Itself. In: Network and Distributed System Security Symposium (NDSS) (2020)
16. Hacker, A.J.: Importance of Web Application Firewall Technology for Protecting Web-based Resources. ICSA Labs an Independent Verizon Business p. 7 (2008)
17. Hubbard, S., Sager, J.: Firewalling the net. BT technology journal **15**(2), 94–106 (1997)
18. IANA functions: Message headers (2020), https://www.iana.org/assignments/message-headers/message-headers.xhtml
19. Imperva: Transparent reverse proxy (2020), https://docs.imperva.com/bundle/v14.1-administration-guide/page/7200.htm
20. Jeremy, D., Hils, A., Kaur, R., Watts, J.: Critical capabilities for cloud web application firewall services (2020), https://www.gartner.com/doc/reprints?id=1-1XO56V9N&ct=191022
21. Keromytis, A.D., Wright, J.L.: Transparent network security policy enforcement. In: USENIX Annual Technical Conference, FREENIX Track. pp. 215–226 (2000)

22. Kettle, J.: Http desync attacks: Request smuggling reborn (2019), https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn
23. Klein, A.: Divide and conquer - http response splitting, web cache poisoning attacks, and related topics (2004), https://dl.packetstormsecurity.net/papers/general/whitepaper_httpresponse.pdf
24. Klein, A.: Http request smuggling in 2020 – new variants, new defenses and new challenges (2020), https://i.blackhat.com/USA-20/Wednesday/us-20-Klein-HTTP-Request-Smuggling-In-2020-New-Variants-New-Defenses-And-New-Challenges-wp.pdf
25. Kogi, E., Kerman, D.: HTTP desync attacks in the wild and how to defend against them (2019), https://www.imperva.com/blog/http-desync-attacks-and-defence-methods/
26. Levine, J.R.: DNS Blacklists and Whitelists. RFC 5782 (Feb 2010). https://doi.org/10.17487/RFC5782, https://rfc-editor.org/rfc/rfc5782.txt
27. Linhart, C., Klein, A., Heled, R., Steve, O.: Http request smuggling (2005), https://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf
28. Lo, J.: Whitelisting for Cyber Security: What It Means for Consumers. Public Interest Advocacy Centre (2011)
29. Ltd., P.: Access control vulnerabilities and privilege escalation (2020), https://portswigger.net/web-security/access-control
30. Mirheidari, S.A., Arshad, S., Onarlioglu, K., Crispo, B., Kirda, E., Robertson, W.: Cached and confused: Web cache deception in the wild. In: 29th USENIX Security Symposium (USENIX Security) (2020)
31. Nguyen, H.V., Lo Iacono, L., Federrath, H.: Your Cache Has Fallen: Cache-Poisoned Denial-of-Service Attack. In: 26th ACM Conference on Computer and Communications Security (CCS) (2019)
32. OWASP Foundation: OWASP top ten web application security risks (2020), https://owasp.org/www-project-top-ten/
33. Pałka, D., Zachara, M.: Learning Web Application Firewall - Benefits and Caveats. In: Tjoa, A.M., Quirchmayr, G., You, I., Xu, L. (eds.) Availability, Reliability and Security for Business, Enterprise and Health Information Systems. pp. 295–308. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
34. Saltzer, J.H., Schroeder, M.D.: The protection of information in computer systems. Proceedings of the IEEE **63**(9), 1278–1308 (1975)
35. Shahzad, A., Hussain, M., Khan, M.N.A.: Protecting from Zero-Day Malware Attacks. Middle-East Journal of Scientific Research **17**(4), 455–464 (2013)
36. Torrano-Gimenez, C., Perez-Villegas, A., Alvarez, G.: A Self-learning Anomaly-Based Web Application Firewall. In: Computational Intelligence in Security for Information Systems. pp. 85–92. Advances in Intelligent and Soft Computing, Springer (2009). https://doi.org/10.1007/978-3-642-04091-7_11