Your Cache Has Fallen: Cache-Poisoned Denial-of-Service Attack

Hoai Viet Nguyen, Luigi Lo Iacono
Data & Application Security Group
Cologne University of Applied Sciences, Germany
{viet.nguyen,luigi.lo_iacono}@th-koeln.de

Hannes Federrath
Security in Distributed Systems Group
University of Hamburg, Germany
federrath@informatik.uni-hamburg.de

ABSTRACT

Web caching enables the reuse of HTTP responses with the aim to reduce the number of requests that reach the origin server, the volume of network traffic resulting from resource requests, and the user-perceived latency of resource access. For these reasons, a cache is a key component in modern distributed systems as it enables applications to scale at large. In addition to optimizing performance metrics, caches promote additional protection against Denial of Service (DoS) attacks.

In this paper we introduce and analyze a new class of web cache poisoning attacks. By provoking an error on the origin server that is not detected by the intermediate caching system, the cache gets poisoned with the server-generated error page and instrumented to serve this useless content instead of the intended one, rendering the victim service unavailable. In an extensive study of fifteen web caching solutions we analyzed the negative impact of the Cache-Poisoned DoS (CPDoS) attack—as we coined it. We show the practical relevance by identifying one proxy cache product and five CDN services that are vulnerable to CPDoS. Amongst them are prominent solutions that in turn cache high-value websites. The consequences are severe as one simple request is sufficient to paralyze a victim website within a large geographical region. The awareness of the newly introduced CPDoS attack is highly valuable for researchers for obtaining a comprehensive understanding of causes and countermeasures as well as practitioners for implementing robust and secure distributed systems.

CCS CONCEPTS

• **Security and privacy → Network security**; **Denial-of-service attacks**; **Web application security**.

KEYWORDS

HTTP; Web Caching; Cache Poisoning; Denial of Service

ACM Reference Format:
Hoai Viet Nguyen, Luigi Lo Iacono and Hannes Federrath. 2019. Your Cache Has Fallen: Cache-Poisoned Denial-of-Service Attack. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3319535.3354215

CCS '19, November 11–15, 2019, London, United Kingdom

1 INTRODUCTION

Contemporary distributed software systems require to scale at large in order to efficiently handle the sheer magnitude of requests stemming, e.g., from human users all over the globe or sensors scattered around in an environment. A common architectural approach to cope with this requirement is to design the system in layers composed of distinct intermediaries. Application-level messages travel through such intermediate systems on their path between a client and a server. Common intermediaries include caches, firewalls, load balancers, document routers and filters.

The caching of frequently used resources reduces network traffic and optimizes application performance and is one major pillar of success of the web. Caches store recyclable responses with the aim to reuse them for recurring client requests. The origin server usually rules whether a resource is cacheable and under which conditions it can be provided by a caching intermediate. Cached resources are unambiguously identified by the cache key that consists most commonly of the HTTP method and the URL, both contained in the request. In case a fresh copy of a requested resource is contained in an intermediate cache, the client receives the cached copy directly from the cache. By this, web caching systems can contribute to an increased availability as they can serve client requests even when the origin server is offline. Moreover, distributed caching systems such as Content Distribution Networks (CDNs) can provide additional safeguards against Distributed DoS (DDoS) attacks.

A general problem in layered systems is the different interpretation when operating on the same message in sequence. As we will discuss in detail in Section 3, this is the root cause for attacks belonging to the family of "semantic gap" attacks [18]. These attacks exploit the difference in interpreting an object by two or more entities. In the context of this paper the problem arises when an attacker can generate an HTTP request for a cacheable resource where the request contains inaccurate fields that are ignored by the caching system but raise an error while processed by the origin server. In such a setting, the intermediate cache will receive an error page from the origin server instead of the requested resource. In other words, the cache can get poisoned with the server-generated error page and instrumented to serve this useless content instead of the intended one, rendering the victim service unavailable. This is why we denoted this novel class of attacks "Cache-Poisoned Denial-of-Service (CPDoS)".

We conduct an in-depth study to understand how inconsistent interpretation of HTTP requests in caching systems and origin servers can manifest in CPDoS. We analyze the caching behavior of error pages of fifteen web caching solutions and contrast them to the HTTP specifications [13]. We identify one proxy cache product and five CDN services that are vulnerable to CPDoS. We find that such semantic inconsistency can lead to severe security consequences as one simple request is sufficient to paralyze a victim

website within a large geographical region requiring only very basic attacker capabilities. Finally, we show that the CPDoS attack raises the paradox situation in which caching services proclaim an increased availability and proper defense against DoS attacks while they can be exploited to affect both qualities.

Overall, we make three main contributions:

(1) We present a class of new attacks, "Cache-Poisoned Denial-of-Service (CPDoS)", that threaten the availability of the web. We systematically study the cases in which error pages are generated by origin servers and then stored and distributed by caching systems. We introduce three concrete attack variations that are caused by the inconsistent treatment of the `X-HTTP-Method-Override` header, header size limits and the parsing of meta characters.
(2) We empirically study the behavior of fifteen available web caching solutions in their handling of HTTP requests containing inaccurate fields and caching of resulting error pages. We find one proxy cache product and five CDN services that are vulnerable to CPDoS. We have disclosed our findings to the affected solution vendors and have reported them to CERT/CC.
(3) We discuss possible CPDoS countermeasures ranging from cache-ignoring instant protections to cache-adhering safeguards.

2 FOUNDATIONS

The web is considered as the world's largest distributed system. With the continuous growing amount of data traveling around the web, caching systems become an important pillar for the scalability of the web [3]. Web caching systems can occur in various in-path locations between client and origin server (see Figure 1). Another distinction point is the classification in private and shared caches. Private caches are only allowed to store and reuse content for one particular user. Client-internal caches of web browsers are one typical example of private cache as they store responses for a dedicated user only. On the other hand, client-side and server-side caches—also known as proxy caches—as well as CDNs deployed in the backbone of the web belong to the family of shared caches, since they provide content for multiple clients. Some web applications may also include a server-internal cache. These caching systems usually support both access policies, i.e., they are able to serve cached resources to multiple users or to one client exclusively.

The cache policy is governed by the content provider by specifying caching declarations defined in RFC 7234 [11]. The web caching standard defines a set of control directives for instructing caches how to store and reuse recyclable responses. The `max-age` and `s-maxage` attributes in the `Cache-Control` response header define, e.g., the maximum duration in seconds that the targeted content is allowed to reside in a cache. The keyword `max-age` is applicable to private and shared caches whereas `s-maxage` only applies to shared web caching systems. Content providers can also use the `Expires` header with an absolute date to define a freshness lifetime. As with `max-age`, the `Expires` is adoptable for private and shared caches. A stored response in a cache is considered as fresh, if it does not exceed the freshness lifetime specified by `max-age`, `s-maxage` and the `Expires` header. If a content provider wishes

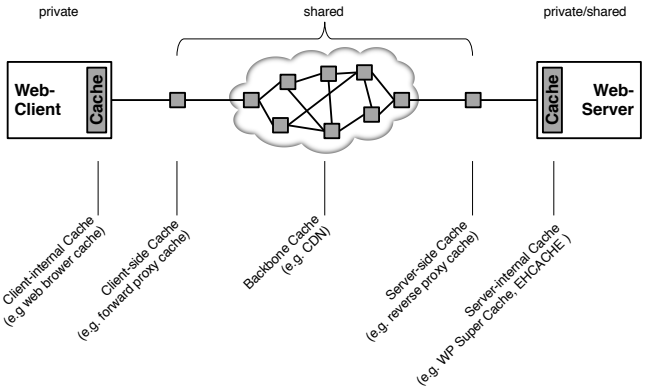

Figure 1: Different types of web caching systems classified by location and resource access policy [31]

to permit a certain content to be saved by private caches only, it adds the `private` directive to the `Cache-Control` header. Content providers which do not want that a certain response is stored and reused by any cache have to include the keyword `no-store` in the `Cache-Control` header. The control directives `must-revalidate`, `proxy-revalidate` and `no-cache` in the `Cache-Control` header instruct how to verify the freshness of a response, in case a content is expired or no freshness lifetime information is available. All mentioned control directives enable a content provider to define caching policies in an explicit manner.

If no explicit caching directive is present in a response, a web caching system may store and reuse responses implicitly when certain conditions are met. One requirement which permits caches to store content implicitly is a response to a GET request. Responses to unsafe methods including POST, DELETE and PUT are not allowed to be cached. Moreover, responses to GET method must contain defined status codes including, e.g., `200 Ok`, `204 No Content` and `301 Moved Permanently`. Here, caches are allowed to derive a freshness lifetime by using heuristics. Many web applications instruct web caching systems to define an implicit freshness lifetime for images, scripts and stylesheets as these file types are considered as static content. Static content refers to data which does not change frequently. Therefore, storing and reusing such resources is considered as best practice for optimizing the performance.

In some cases, it is also very useful for content providers to cache certain error messages. For instance, the status code `404 Not Found`, which indicates that the origin server does not have a suitable representation for the requested resource, is permitted to be cached implicitly. The `405 Method Not Allowed` declaring the request action is not supported for the targeted resource can be cached implicitly as well.

3 SECURITY THREATS IN WEB CACHING SYSTEMS

Using web caching systems provides many advantages in terms of optimizing communication and application performance. However, much work has shown that web caches can also be exploited to affect the privacy and reliability of applications. Web cache poisoning attacks, e.g., are a serious threat that has been emerging

over the past years. Amongst them is the *request smuggling* [24] attack which occurs when the web caching system and the origin server do not strictly conform to the policies specified by RFC 7234. In this particular attack, the attacker can send a request with two `Content-Length` headers to impair a shared cache. Even though the presence of two `Content-Length` headers is forbidden as per RFC 7234, some HTTP engines in caches and origin servers still parse the request. Due to the duplicate headers, the malformed request is able to confuse the origin server and the cache so that a harmful crafted response can be injected to the web caching system. This malicious response is then reused for recurring requests.

The *host of troubles* [7] attack is another vulnerability targeting shared caches. As with the previous attack, it exploits a violation of the web caching standard that gets interpreted differently by the involved system layers. Here, the attacker constructs a request with two `Host` headers. These duplicate headers induce a similar misbehavior in the cache and origin server as the request smuggling attack. Likewise, a malicious response is injected to poison the cache.

Another attack that targets to poison web caches is the *response splitting* [23] attack. Unlike the two aforementioned vulnerabilities, where a flaw in the shared cache itself is one reason why the attack is successful, the response splitting attack exploits a parsing issue in the origin server only. Here, an attacker utilizes the fact that the HTTP engine of the origin server does not escape or block line breaks when replaying a request header value in the corresponding response header. A malicious client can exploit this by dividing the response in two responses. The aim of this attack is to poison the intermediate cache with the malicious content contained in the second response.

James Kettle [22] presented a set of cache poisoning attacks which result from a misbehavior in web application frameworks and content management systems respectively. With the introduced techniques, James Kettle was able to compromise shared web caching systems of well-known companies.

All introduced attacks aim at poisoning shared caches with malicious content that gets served by the victim caches for recurring requests of benign clients. Private caches such as the web browser cache are not affected by the mentioned attacks. However, browser caches are not immune to this class of attacks. Jia et al. [19] present browser cache poisoning (BCP) attacks. In their study they find that many desktop web browsers are susceptible to BCP attacks.

The *web cache deception* [15] attack targets to poison a shared cache with sensitive content. Here, the attacker exploits a RFC 7234 violation of a shared cache which still stores responses even though it is prohibited. In combination with an issue in the request routing of the origin server, the author was able to retrieve account information of third parties out of the cache.

Triukose et al. [39] showed another attack vector that utilizes web caching systems to paralyze a web application. Unlike the presented threats, this attack does not intend to poison a cache with harmful content or to steal sensitive data. The goal of Triukose et al. was to provoke a DoS attack with the aid of a mounted CDN. The authors utilized the infrastructure of a CDN, which comprises of many collaborating edge cache servers. With the use of a random string appended to the URL query, Triukose et al. were able to bypass any edge cache servers so that the CDN forwards every request to the origin server. To create a DoS attack, the authors send multiple requests with different random query strings to all edge cache servers within the CDN. As the edge cache servers forward all of these requests to the origin server, the huge amount of requests reaching the origin server generates a high workload with the consequence that the web application cannot process any further legitimate request.

The root cause of almost all of the presented attacks lies in the different interpretation of HTTP messages by two or more distinct message processing entities, which is known as the *semantic gap* [18]. Vulnerabilities stemming from the semantic gap are manifold [7, 23, 37]. In relation to web caches the request smuggling, host of troubles and response splitting attacks exploit this gap between a cache and an origin server. Here, a discrepancy in parsing duplicate headers or line breaks leads to cache poisoning.

In the next section we introduce a new class of attacks against web caches, the Cache-Poisoned Denial-of-Service (CPDoS) attack. It exploits the semantic gap between a shared cache and a origin server for poisoning the cache with error pages. As a consequence, the cache distributes error pages instead of the legitimate content after being poisoned. Users perceive this as unavailable resources or services. In contrast to the DDoS attack introduced by Triukose et al., CPDoS require only very basic attack skills and resources.

4 POISONING WEB CACHES WITH ERROR PAGES

The general attack idea is to exploit the semantic gap in two distinct HTTP engines—one contained in a shared cache and the other in an origin server. More specifically, the baseline of the newly introduced variant of web cache poisoning takes advantage of the circumstance that the deployed caching system is more lax or focused in processing requests than the origin server (see Figure 2). An attacker can make use of this discrepancy by including a customized malicious header or multiple harmful headers in the request. Such headers are usually forwarded without any changes to the origin server. As a consequence, the attacker crafted request runs through the cache without any issue, while the server-side processing results in an error. Henceforth, the server's response is a respective error, which will be stored and reused by the cache for recurring requests. Each benign client making a subsequent GET request to the infected URL will receive a stored error message instead of the genuine resource form the cache.

It is worth noting that one simple request is sufficient to replace the genuine content in the cache by an error page. This means that such a request remains below the detection threshold of web application firewalls (WAFs) and DDoS protection means in particular, as they scan for large amounts of irregular network traffic.

The consequences for the web application depend on the content being illegitimately replaced with error pages. It will always affect the service's availability—either parts of it or entirely. The most harmless CPDoS renders images or style resources unavailable. This influences the visual appearance of parts of the application. In terms of functionality it is still working, however. More serious attacks targeting the start page or vital script resources can render the entire web application inaccessible instead. Moreover, CPDoS can be exploited to block, e.g., patches or firmware updates distributed

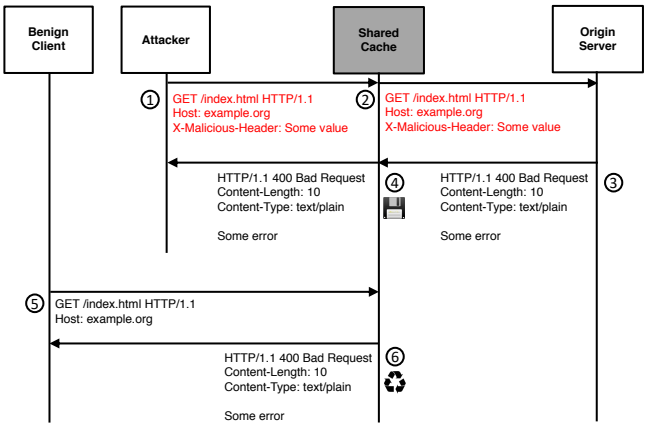

Figure 2: General construction of the Cache-Poisoned Denial-of-Service (CPDoS) attack

via caches, preventing vulnerabilities in devices and software from being fixed. Attackers can also disable important security alerts or messages on mission-critical websites such as online banking or official governmental websites. Imagine, e.g., a situation in which a CPDoS attack prevents alerts about phishing emails or natural catastrophes from being displayed to the respective user.

When considering the low efforts for attackers, the high probability of success, the low chance of being detected and the relatively high consequences of a DoS then the introduced CPDoS attack poses a high risk. Hence, it is worthwhile investigating under which conditions CPDoS attacks can occur in the wild. For this reasons we first compiled a complete overview on cacheable error codes as specified in relevant RFCs [16], [25], [32], [9], [8], [34], [13], [11], [4] and [5] (see Table 1). Moreover, we analyzed whether popular proxy caches as well as CDNs do store and reuse error codes returned from the origin server. This exploratory study has been conducted with the approach of Nguyen et al. [30, 31]. They provide a freely available cache testing tool for analyzing web browser caches, proxy caches and CDNs in a systematically manner. The cache testing tool also offers a test suite containing 397 test cases that can be customized by a test case specification language. We extended the suite by adding new tests for evaluating the caching of responses containing error status codes. In our study we concentrated on the five well-known proxies caches Apache HTTP Server (Apache HTTPD) v2.4.18, Nginx v1.10.3, Varnish v6.0.1, Apache Traffic Server (Apache TS) v8.0.2 and Squid v3.5.12 as well as the CDNs Akamai, CloudFront, Cloudflare, Stackpath, Azure, CDN77, CDNSun, Fastly, KeyCDN and G-Core Labs.

Even though the cacheability of error codes are well-defined by the series of RFC specifications given above, our analysis reveals that some web caching systems violates some of these policies. For instance, CloudFront and Cloudflare do store and reuse error messages such as `400 Bad Request`, `403 Forbidden` and `500 Internal Server Error` although being not permitted. The violation of web caching policies is a severe issue and needs to be taken into account by content providers and web caching system vendors. Recent publications have revealed that non-adherence may otherwise lead to caching vulnerabilities [7, 15, 24]. Following these observations, we investigated further in order to discover vulnerable constellations. We were able to identify three concrete instantiations of the general CPDoS attack that we present in the following subsections.

4.1 HTTP Method Override (HMO) Attack

The HTTP standard [13] defines a set of request methods for the client to indicate the desired action to be performed for a given resource. GET, POST, DELETE, PUT and PATCH are arguably the most used HTTP methods in web applications and REST-based web services [36] in particular. Some intermediate systems such as proxies, load balancer, caches or firewalls, however, only support GET and POST. This means DELETE, PUT and PATCH requests are simply blocked. To circumvent this restriction many REST-based APIs or web frameworks provide auxiliary headers such as `X-HTTP-Method-Override`, `X-HTTP-Method` or `X-Method-Override` for passing through an unrecognized HTTP method. These headers will usually be forwarded by any intermediate systems. Once the request reaches the server, a method override header instructs the web application to replace the method in the request line with the one in the method overriding header value.

These method override headers are very useful in scenarios when intermediate systems block distinct HTTP methods. However, if a web application supports such a header and also uses a shared web caching system, a malicious client can exploit this semantic gap for performing a CPDoS attack. In a typical HTTP Method Override (HMO) attack flow, a malicious client crafts a GET request including an HTTP method overriding header as shown in Figure 3.

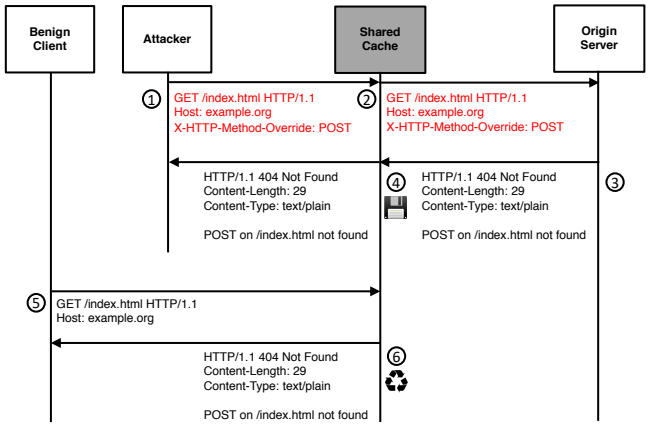

Figure 3: Flow and example construction of the HTTP Method Override (HMO) attack

A CDN or reverse proxy cache interprets the request in Figure 3 as a benign GET request targeting http://example.org/index.html. Hence, it forwards the request with the `X-HTTP-Method-Override` header to the origin server. The endpoint, however, interprets this request as a POST request, since the `X-HTTP-Method-Override` header instructs the server to replace the HTTP method in the request line with the one contained in the header. Accordingly, the web application returns a response based on POST. Let's assume that the target web application does not implement any POST

Legend: ✓ cacheable status code according to HTTP Standard, ● stored by web caching system, ○ not stored by web caching system, ■ storing not cacheable status code

Error Code	Cacheable	Apache HTTPD	Apache TS	Nginx	Squid	Varnish	Akamai	Azure	CDN77	CDNSun	Cloudflare	CloudFront	Fastly	G-Core Labs	KeyCDN	Stackpath
400 Bad Request	–	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○
401 Unauthorized	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
402 Payment Required	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
403 Forbidden	–	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○
404 Not Found	✓	○	○	○	○	●	●	○	●	○	●	●	●	○	○	○
405 Method Not Allowed	✓	○	○	○	○	○	●	○	○	○	○	●	○	○	○	○
406 Not Acceptable	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
407 Proxy Authentication Required	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
408 Request Timeout	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
409 Conflict	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
410 Gone	✓	○	●	○	○	●	●	●	○	○	●	●	●	○	○	○
411 Length Required	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
412 Precondition Failed	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
413 Payload Too Large	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
414 Request-URI Too Long	✓	○	○	○	○	●	○	○	○	○	○	●	○	○	○	○
415 Unsupported Media Type	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
416 Requested Range Not Satisfiable	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
417 Expectation Failed	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
418 I'm a teapot	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
421 Misdirected Request	✓	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
422 Unprocessable Entity	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
423 Locked	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
424 Failed Dependency	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
426 Upgrade Required	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
428 Precondition Required	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
429 Too Many Requests	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
431 Request Header Fields Too Large	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
444 Connection Closed Without Response	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
451 Unavailable For Legal Reasons	✓	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
499 Client Closed Request	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
500 Internal Server Error	–	○	○	○	○	○	○	■●	○	○	○	■●	○	○	○	○
501 Not Implemented	✓	○	○	○	○	○	○	●	○	○	○	●	○	○	○	○
502 Bad Gateway	–	○	○	○	○	○	○	■●	○	○	○	■●	○	○	○	○
503 Service Unavailable	–	○	○	○	○	○	○	■●	○	○	○	■●	○	○	○	○
504 Gateway Timeout	–	○	○	○	○	○	○	■●	○	○	○	■●	○	○	○	○
505 HTTP Version Not Supported	–	○	○	○	○	○	○	■●	○	○	○	○	○	○	○	○
506 Variant Also Negotiates	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
507 Insufficient Storage	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
508 Loop Detected	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
510 Not Extended	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
511 Network Authentication Required / Status Code and Captive Portals	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
599 Network Connect Timeout Error	–	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○

Table 1: Overview of cacheable error status codes according to [4, 5, 8, 9, 11, 13, 16, 25, 32, 34] and empirical study results showing whether the status codes are cached by the analyzed web caching systems

endpoint for /index.html. In such a case, web frameworks usually returns an error message, e.g., the status code `404 Not Found` or `405 Method Not Allowed`. The shared cache assigns the returned response with the error code to the GET request targeting http://example.org/index.html. Since the status codes `404 Not Found` and `405 Method Not Allowed` are cacheable according to the HTTP Caching RFC 7231 as shown in Table 1, caches store and reuse this error response for recurring requests. Each benign client making a subsequent GET request to http://example.org/index.html receives the cached error message instead of the legitimated web application's start page.

4.2 HTTP Header Oversize (HHO) Attack

The HTTP standard does not define any size limit for request headers. Hence, intermediate systems, web servers and web frameworks specify their own limit. Most web servers and proxy caches provide a request header limit of about 8,000 bytes in order to avoid security threats such as *request header overflow* [26] or *ReDoS* [38] attacks. However, there are also intermediate systems, which specify a limit larger than 8,000 bytes. For instance, the Amazon CloudFront CDN allows up to 24,713 bytes. In an exploratory study we gathered the default HTTP request header limits deployed by various HTTP engines and cache systems (see Table 3).

This semantic gap in terms of different request header size limits can be exploited to conduct a CPDoS attack. To execute an HTTP Header Oversize (HHO) attack, a malicious client needs to send a GET request including a header larger than the limit of the origin server but smaller than the one of the cache. To do so, an attacker has two options. First, she crafts a request header with many malicious headers. The other option is to include one single header with an oversized key or value as shown in Figure 4.

The web caching system forwards this request including the oversized header to the endpoint, since the header size is under the limit of the intermediary. The web server, however, blocks this request and returns an error page, as the request exceeds the header size limit. This returned error page is stored and will be reused for equivalent requests.

4.3 HTTP Meta Character (HMC) Attack

The HTTP Meta Character (HMC) works similar to the HHO attack. Instead of sending an oversized header, this attack tries to bypass a cache with a request header containing a harmful meta character. Meta characters can be e.g. control characters such as the line break/carriage return (`\n`), line feed (`\r`) or any other Unicode control characters. As the `\n` and `\r` characters are used by the response

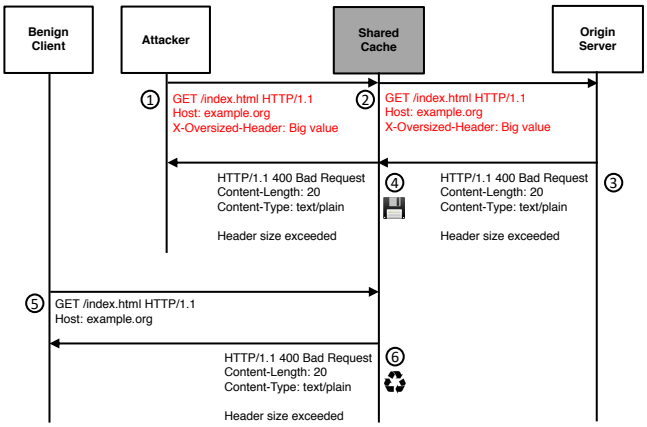

Figure 4: Flow and example construction of the HTTP header oversize (HHO) attack

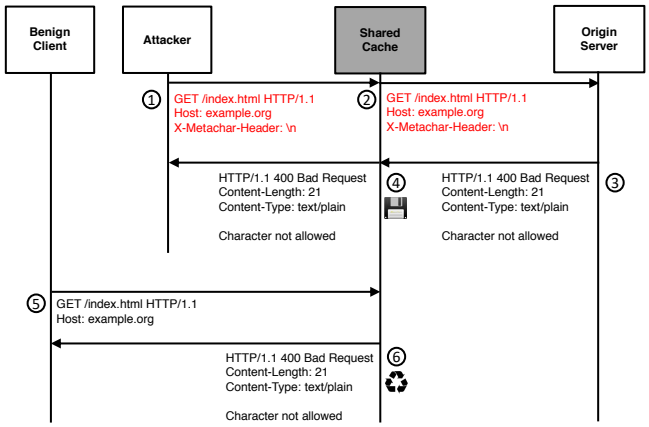

Figure 5: Flow and example construction of the HTTP Meta Character (HMC) attack

splitting attack to poison a cache, some HTTP implementations block requests containing these symbols.

HTTP implementations, which drop such characters, mostly return an error message signaling that they do not parse this request. However, there are some cache intermediaries which do not care about certain control characters. They simply forward the request including the meta character to the origin server which return an error code. The resulting error page is then stored and reused by the cache. This constellation can be exploited by a malicious client to conduct another form of CPDoS attack. We declare this vulnerability as HTTP Meta Character (HMC) attack. To do so, the attacker crafts a request with a meta character, e.g. `\n`, as shown in Figure 5. The goal of this example attack in is to fool the origin server into believing that it is attacked by a response splitting request. As with the previously presented vulnerabilities, the HMO request traverses the cache without any issues. Once the request reaches the endpoint, it is blocked and an according error page is returned, since the web server is aware of the implications regarding suspicious characters such as `\n`. This error message is then stored and recycled by the corresponding web caching system.

5 PRACTICABILITY OF CPDOS ATTACKS

In order to explore the existence of CPDoS weaknesses in the wild, we conducted a series of experiments. A crucial prerequisite for a potential CPDoS vulnerability is a web caching system that stores and reuses error pages produced by the origin server. Table 1 highlights that Varnish, Apache TS, Akamai, Azure, CDN77, Cloudflare, CloudFront and Fastly do so. Based on these findings, we conducted three experiments—one for each introduced CPDoS variant—to examine whether these intermediate systems are vulnerable to CPDoS attacks.

5.1 Experiments Setup

The first step to analyze whether CPDoS vulnerabilities exist in practical environments is to figure out vulnerable HTTP implementations which are utilized as the origin server. HTTP implementations on the origin server can be diverse systems including, e.g., reverse proxies, web servers, web frameworks, cloud services or other intermediate systems as well as another cache.

In our first experiment, we analyzed the method override header support in web frameworks. Additionally, we also evaluated what error page is returned when sending a method override header containing an HTTP method which is not implemented by corresponding resource endpoint. Based on the findings in Table 1 where we know what error page is stored by what web caching systems, we inferred what web framework in combination with what web caching systems might be vulnerable to HMO attacks. For this empirical analysis we chose 13 web frameworks based on the most popular programming languages according to IEEE Spectrum [17]. The analyzed collection of web frameworks includes ASP.NET v2.2, BeeGo v1.10.0, Django v2.1.7, Express.js v.4.16.4, Flask v1.0.2, Gin v1.3.0, Laravel v5.7, Meteor.js v1.8, Rails v5.2.2, Play Framework 1 (Play 1) v1.5.1, Play Framework 2 (Play 2) v2.7, Spring Boot v2.1.2 and Symfony v4.2.

The second experiment investigated the request header size limits of the web caching systems in Table 1 as well as the 13 web frameworks. As the web frameworks ASP.NET and Spring Boot requires an underlying web server to be deployed in production mode, we additionally also evaluate the request header limits of Microsoft Internet Information Services (IIS) v10.0.17763.1 and Tomcat v9.0.14. Moreover, we also evaluated popular cloud services including Amazon S3, Github Pages, Gitlab Pages, Google Storage and Heroku. As with the first experiment, we also tested which error code is returned when the request header size limit is exceeded. With these findings we figured out what HTTP implementations in conjunction with what web caching systems are potentially vulnerable to HHO attacks.

The last experiment evaluated the feasibility of HMC attacks. Here, we evaluated the handling of meta characters in all mentioned web caching systems, web frameworks, web servers and cloud services. To test as many meta characters as possible we collected as list of 520 potentially irritating strings. This collection contains control, special, international and other unicode characters as well as strings comprising attack vectors including cross site scripting (XSS), SQL injections and remote execution attacks. The goals of

this study was to analyze what characters and strings are blocked, sanitized and processed or forwarded without any issues. Moreover, we also evaluated what error page is triggered when a character or string is blocked. Based on our findings we were able to conclude what characters and what symbols need to be send to what constellation of HTTP engine and web caching system to induce an HMC attack.

5.2 Feasibility of HMO attacks

Table 2 shows the results of the first experiment. It highlights that Symfony, Laravel and Play 1 support method override headers by default. Django and Express.js instead do not consider method override headers by default, but provide plugins to add this feature. Flask does not offer any plugin for the integration of method override headers, but provides an official tutorial how to enable it [14]. Table 2 also points out what error code is returned when the web framework receives a method override header with an action that is not implemented by the addressed resource endpoint.

Even though the web frameworks with a method overriding header support return cacheable error codes, we observed that only Play 1 and Flask are vulnerable to HMO CPDoS attacks. However, both web frameworks can only be affected if Fastly, Akamai, Cloudflare, CloudFront, CDN77 and Varnish are used as intermediate cache. The reason why these web frameworks are vulnerable lies in the fact that Play 1 and Flask do perform an HTTP method change for GET as well as POST requests in case an HTTP method override header is present. Laravel, Symfony and the plugins for Django and Express.js are not vulnerable to HMO CPDoS, since they ignore HTTP method override headers in GET requests and restrict themselves to transform the method for POST requests only. Attackers cannot poison the tested web caching systems with a POST request, since responses to POST requests are not stored by any of them.

Malicious clients can attack web applications implemented with the Play 1 by sending a GET request with the method override header including, e.g., POST as value. If the corresponding resource endpoint does not implement any functionality for POST, then the web framework returns the error code `404 Not Found`. Akamai, Fastly, CDN77, Cloudflare, CloudFront and Varnish cache this status code by default (see Table 1). Flask is also vulnerable to HMO CPDoS attacks, if the support of HTTP method override headers is implemented with the official tutorial of the web framework's website. However, HMO attacks are only possible, if Akamai and CloudFront are utilized as CDN, since Flask returns the status code `405 Method Not Allowed`. Akamai and CloudFront are the only analyzed web caching systems, which store and reuse error pages with this code.

5.3 Feasibility of HHO attacks

Table 3 depicts the results of our study on request header size limits. If available, it moreover lists the request header size limit specified in the documentation of the corresponding HTTP implementation. Note, that we omit the web frameworks ASP.NET, Django, Flask, Laravel, Rails, Symfony and Spring Boot in this table, as we found out that the request header limits depend on the used web server and deployment environment.

Our obtained results reveal many varieties in terms of request header size limits among the HTTP implementations. The evaluation shows that CloudFront provides a request header size limit, which is much higher than the one of the many other HTTP implementations we tested. Moreover, Amazon's CDN also caches the error code `400 Bad Request` by default (see Table 1), which is triggered by most of the HTTP implementations when the request header size limit is exceeded. Hence, in our experiments we figured out that when using CloudFront as CDN any HTTP implementation that has a request header size limit lower than CloudFront and returns the status code `400 Bad Request` if the limit is exceeded is vulnerable to HHO CPDoS atacks. For instance, the web caching systems Apache HTTPD and Nginx, which can also be used as web server or reverse proxy provide a lower request header size limit than CloudFront.

Besides the fact that Apache HTTPD and Nginx are amongst the most used web servers according to a survey of Netcraft [28], both systems are often deployed with other intermediate systems. When using one of these HTTP implementations in conjunction with CloudFront, these systems can be affected by an HHO CPDoS attack. This also means if Apache HTTPD and Nginx is configured as intermediate reverse proxy in front of other web applications, then these systems are vulnerable to HHO CPDoS as well. Moreover, Apache HTTPD and Nginx are often utilized as web server and deployment environment for web frameworks such as Rails, Django, Flask, Symfony and Lavarel. All these web frameworks are vulnerable to HHO CPDoS likewise if they are deployed with Apache HTTPD or Nginx. Spring Boot and ASP.NET can also be affected by HHO CPDoS attacks, as both web frameworks require a web server in production mode. Spring Boot can be deployed with Tomcat and ASP.NET can use IIS as the underlying deployment environment. Tomcat and IIS have request header size limits lower than CloudFront. Both web servers return the error `400 Bad Request` for oversized header likewise. The cloud service Heroku is another deployment platform for web frameworks. It supports, e.g., Django, Flask, Laravel, Rails, Laravel and Symfony. As Heroku provides a request header size limit lower than CloudFront, web applications using the cloud service in conjunction with the CDN can be vulnerable as well. Other HTTP implementations which can be affected by HHO CPDoS attacks when using CloudFront as CDN are Play 2 as well as the cloud services Amazon S3, Github Pages and Heroku. Play 1 is also vulnerable to HHO CPDoS attacks, even though it does not return an error page when the request header size limit is exceeded. The web framework does not return any response if it receives an oversized header. Here, the TCP socket remains open until the web application shuts down. If CloudFront notices such an idle communication channel, then the CDN returns the error code `502 Bad Gateway`. This error message is stored and reused for recurring requests likewise. According to our experiments, Google storage in conjunction with CloudFront is not vulnerable to HHO CPDoS although the cloud service has lower request header size limit than the CDN. Google storage returns the error code `413 Payload Too Large` for oversized headers and this error message is not cached by any of the analyzed web caching systems. Table 3 also contains a result obtained when using Nginx with the WAF plugin ModSecurity. In such a configuration, conducting a successful HHO CPDoS attack is even easier as without

Legend: ○ must be implemented manually, ● by default, ◐ not by default but by extension

Web framework	Programming lang.	Method overriding support	Error code when method not implemented
Rails	Ruby	○	undefined
Django	Python	◐	405
Flask	Python	◐	405
Express.js	JavaScript	◐	405
Meteor.js	JavaScript	○	undefined
BeeGo	Go	○	undefined
Gin	Go	○	undefined
Play 1	Java	●	404
Play 2	Java/Scala	○	undefined
Spring Boot	Java	○	undefined
Symfony	PHP	●	405
Lavarel	PHP	●	405
ASP.NET	C#	○	undefined

Table 2: HTTP method overriding headers support of tested web frameworks

	HTTP implementation	Documented limit	Tested limit	Limit exceed error code
CDN	Akamai	undefined	32,760 bytes	No Response
	Azure	undefined	24,567 bytes	400
	CDN77	undefined	16,383 bytes	400
	CDNSun	undefined	16,516 bytes	400
	Cloudflare	undefined	≈ 32,395 bytes	400
	Cloudfront	20,480 bytes	≈ 24,713 bytes	494
	Fastly	undefined	69,623 bytes	No Response
	G-Core Labs	undefined	65,534 bytes	400
	KeyCDN	undefined	8,190 bytes	400
	StackPath	undefined	≈ 85,200 bytes	400
HTTP engine	Apache HTTPD	8,190 bytes	8,190 bytes	400
	Apache HTTPD + ModSecurity	undefined	8,190 bytes	400
	Apache TS	131,072 bytes	65,661 bytes	400
	Nginx	undefined	20,584 bytes	400
	Nginx + ModSecurity	undefined	8,190 bytes	400
	IIS	undefined	16,375 bytes	400, (404)
	Squid	65,536 bytes	65,527 bytes	400
	Tomcat	undefined	8,184 bytes	400
	Varnish	8,192 bytes	8,299 bytes	400
Cloud Service	Amazon S3	undefined	≈ 7,948 bytes	400
	Github Pages	undefined	8,190 bytes	400
	Gitlab Pages	undefined	>500,000 bytes	undefined
	Google Cloud Storage	undefined	16,376 bytes	413
	Heroku	8,192 bytes	8,154 bytes	400
Web Framework	BeeGo	undefined	>500,000 bytes	undefined
	Express.js	undefined	81,867 bytes	No Response
	Gin	undefined	>500,000 bytes	undefined
	Meteor.js	undefined	81,770 bytes	400
	Play 1	undefined	8,188 bytes	No Response
	Play 2	8,192 bytes	8,319 bytes	400

Table 3: Request header size limits of HTTP implementations

the security extension. The tested request header limit of Nginx is around 20,000 bytes but when ModSecurity is added to both systems, it reduces the restriction to 8,190 bytes. Even though the usage of ModSecurity should actually avoid web application attacks such as DoS, it eases to conduct an HHO CPDoS attack in this case.

As mentioned before, IIS and web frameworks such as APS.NET running on this web server are vulnerable to HHO CPDoS attacks when using CloudFront as CDN. However, in certain circumstances, they might also be vulnerable when Akamai, Fastly, CDN77, Cloudflare and Varnish are utilized. The IIS web server provides an option to set a size limit for a distinct request header. Some web applications require such a configuration option to block, e.g., an oversized `Cookie` header. If this restriction is defined for a request header and this limit is exceeded, then the web server return the error code `404 Not Found`. This error message is cached by Akamai, Fastly, CDN77, CloudFront, Cloudflare and Varnish.

5.4 Feasibility of HMC attacks

Table 4 shows the results of our third experiments where we analyzed the handling of strings containing meta characters. For the sake of readability, we only list the characters and strings that are blocked or sanitized by at least one of the tested HTTP implementations. Moreover, we omit the web frameworks ASP.NET, Django, Flask, Laravel, Spring Boot and Symfony in this table, since the handling of meta characters depends on the used web server and deployment environment.

The evaluation highlights that the many analyzed systems consider control characters as a threat. Suspicious characters or strings are either blocked by the denoted error code or are sanitized from the request header. However, the handling of meta strings and characters are very diverse. For instance, CloudFront blocks the character `\u0000` and sanitizes `\n`, `\v`, `\f`, `\r`, but forwards other control characters such as `\a`, `\b` and `\e` without modifying them. If Apache HTTPD, IIS or Varnish is used with CloudFront, then the corresponding systems block the forwarded header containing forbidden characters with the status code `400 Bad Request`. CloudFront stores such an error message. This means when using CloudFront as CDN, all tested HTTP implementations, which blocked harmful strings and characters that are not rejected or sanitize by CloudFront, are vulnerable to HMC CPDoS attacks. Besides Apache HTTPD, IIS and Varnish, this includes Github Pages, Gitlab Pages, BeeGo, Gin, Meteor.js and Play 2. Express.js is vulnerable to HMC CPDoS attacks as well, even though it does not block any tested string by an error code. The issue here is similar to

Legend: ○ processed/forwarded without error and sanitization

Meta character in request header	Akamai	Azure	CDN77	CDNSun	Cloudflare	Cloudfront	Fastly	G-Core Labs	KeyCDN	Stackpath
\u0000	400	400	400	400	400	400	No Response	400	400	Sanitized
\u0001 ... \u0006	○	400	Sanitized	○	○	○	400	○	○	○
\a	○	400	Sanitized	○	○	○	400	○	○	○
\b	○	400	Sanitized	○	○	○	400	○	○	○
\t	○	○	○	○	○	○	○	○	○	○
\n	○	400	Sanitized	Sanitized	Sanitized	Sanitized	Sanitized	Sanitized	○	Sanitized
\v	○	400	Sanitized	○	○	Sanitized	400	○	○	Sanitized
\f	○	400	Sanitized	○	○	Sanitized	400	○	○	Sanitized
\r	○	400	Sanitized	○	Sanitized	Sanitized	400	Sanitized	○	Sanitized
\u000e ... \001f, \u007f	○	400	Sanitized	○	○	○	400	○	○	○
Multiple Unicode control character (e.g.\u0001\u0002)	○	400	Sanitized	○	○	○	400	○	○	○
`(){0;}; touch /tmp/blns.shellshock1.fail;`	○	○	○	○	403	○	○	○	○	○
`() { _; } >_[$($())] { touch /tmp/blns.shellshock2.fail; }`	○	○	○	○	403	○	○	○	○	○

Meta character in request header	Apache HTTPD + (ModSecurity)	Apache TS	Nginx + (ModSecurity)	IIS	Tomcat	Squid	Varnish	Amazon S3	Google Storage
\u0000	400	400	400	400	○	○	400	○	○
\u0001 ... \u0006	400	○	○	400	○	○	400	○	○
\a	400	○	○	400	○	○	400	○	○
\b	400	○	○	400	○	○	400	○	○
\t	○	○	○	400	○	○	400	○	○
\n	400	○	Sanitized	○	○	○	Sanitized	○	○
\v	400	○	○	400	○	○	400	○	○
\f	400	○	○	400	○	○	400	○	○
\r	400	○	○	400	○	○	400	○	○
\u000e ... \001f, \u007f	400	○	○	400	○	○	400	○	○
Multiple Unicode control character (e.g.\u0001\u0002)	400	○	○	400	○	○	400	○	○
`(){0;}; touch /tmp/blns.shellshock1.fail;`	○	○	○	○	○	○	○	○	○
`() { _; } >_[$($())] { touch /tmp/blns.shellshock2.fail; }`	○	○	○	○	○	○	○	○	○

Meta character in request header	Github Pages	Gitlab Pages	Heroku	Beego	Express.js	Gin	Meteor	Play 1	Play 2
\u0000	No Response	400	○	400	○	400	400	○	400
\u0001 ... *\u0006*	400	400	○	400	○	400	400	○	400
\a	400	400	○	400	○	400	400	○	400
\b	400	400	○	400	○	400	400	○	400
\t	400	○	○	○	○	○	○	○	○
\n	400	○	400	○	○	○	○	○	○
\v	400	400	○	400	○	400	400	○	400
\f	400	400	○	400	○	400	400	○	400
\r	400	400	○	400	○	400	○	○	400
\u000e ... \001f	400	400	○	400	○	400	400	○	400
\u0007f	400	400	○	400	○	400	400	○	○
Multiple Unicode control character (e.g.\u0001\u0002)	400	400	○	400	No Response	400	No Response	○	400
`(){0;}; touch /tmp/blns.shellshock1.fail;`	○	○	○	○	○	○	○	○	○
`() { _; } >_[$($())] { touch /tmp/blns.shellshock2.fail; }`	○	○	○	○	○	○	○	○	○

Table 4: Meta string handling in request header of HTTP implementations

the problem of oversized header in Play 1. When sending a request header with multiple control characters Express.js does not reply at all. Accordingly, CloudFront returns the error message `502 Bad Gateway` to the client. This error code is also stored and reused for subsequent requests.

5.5 Consolidated Review of Analysis Results

Based on our findings of all three experiments, we detected many CPDoS attack vectors in various different combinations of web caching systems and HTTP implementations. Most of the attacks are executable on CloudFront as shown in Table 5. This overview summarizes what pair of web caching system and HTTP implementation is vulnerable to what CPDoS attack. The experiments' results show that web applications using CloudFront are highly vulnerable to CPDoS attacks, since the CDN caches the error code `400 Bad Request` by default. Many server-side HTTP implementations return this error message when sending a request with an oversized header or meta characters. The likelihood to be affected by CPDoS attacks when utilizing the other analyzed caches including Varnish, Akamai, CDN77, Cloudflare or Fastly is rather lower. These web caching systems do store the error code `404 Not Found` but not `400 Bad Request`. The caching of error pages with status code `404 Not Found` is a proper and compliant approach for optimizing website performance. In this case, there is no malfunction in Varnish, Akamai, CDN77, Cloudflare and Fastly. The reason for a successful CPDoS attack lies in the fact that, Play 1 and Microsoft IIS allows to provoke `404 Not Found` error pages on resource endpoints which do not return an error message when sending a benign request.

5.6 Practical Impact

In the first step to estimate the practical impact of CPDoS attacks, we determined the amount of websites that use one of the vulnerable web caching systems and HTTP implementations listed in Table 5. Our approach to find vulnerable real world websites is to inspect the response header.

Many HTTP implementations append informational headers to the response for declaring that a message is processed by this entity. For instance, CloudFront includes the values `Hit from CloudFront` or `Miss from CloudFront` to the `x-cache` header and Microsoft IIS adds the string `Microsoft-IIS` to the `Server` header. By means of this information an attacker can unambiguously detect what cache or what server-side HTTP implementation is used by the target web application respectively. Based on this approach, we analyzed the websites of the U.S. Department of Defense (DoD)[1] and the Alexa

[1] https://dod.defense.gov/About/Military-Departments/DoD-Websites/

Legend: ○ no CPDoS attack dectected

Apache HTTPD	Apache TS	Nginx	Squid	Varnish	Akamai	Azure	CDN77	CDNSun	Cloudflare	CloudFront	Fastly	G-Core Labs	KeyCDN	StackPath	Web caching system / Origin server HTTP implemenation
○	○	○	○	○	○	○	○	○	○	HHO, HMC	○	○	○	○	Apache HTTPD + (ModSecurity)
○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	Apache TS
○	○	○	○	○	○	○	○	○	○	HHO	○	○	○	○	Nginx + (ModSecurity)
○	○	○	○	(HHO)	(HHO)	○	(HHO)	○	(HHO)	HHO, HMC	(HHO)	○	○	○	IIS
○	○	○	○	○	○	○	○	○	○	HHO	○	○	○	○	Tomcat
○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	Squid
○	○	○	○	○	○	○	○	○	○	HHO, HMC	○	○	○	○	Varnish
○	○	○	○	○	○	○	○	○	○	HHO	○	○	○	○	Amazon S3
○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	Google Cloud Storage
○	○	○	○	○	○	○	○	○	○	HHO, HMC	○	○	○	○	Github Pages
○	○	○	○	○	○	○	○	○	○	HMC	○	○	○	○	Gitlab Pages
○	○	○	○	○	○	○	○	○	○	HHO	○	○	○	○	Heroku
○	○	○	○	(HHO)	(HHO)	○	(HHO)	○	(HHO)	(HHO), (HMC)	(HHO)	○	○	○	ASP.NET
○	○	○	○	○	○	○	○	○	○	HMC	○	○	○	○	BeeGo
○	○	○	○	○	○	○	○	○	○	(HHO), (HMC)	○	○	○	○	Django
○	○	○	○	○	○	○	○	○	○	HMC	○	○	○	○	Express.js
○	○	○	○	○	(HMO)	○	○	○	○	HMO, (HHO), (HMC)	○	○	○	○	Flask
○	○	○	○	○	○	○	○	○	○	HMC	○	○	○	○	Gin
○	○	○	○	○	○	○	○	○	○	(HHO), (HMC)	○	○	○	○	Laravel
○	○	○	○	○	○	○	○	○	○	HMC	○	○	○	○	Meteor.js
○	○	○	○	HMO	HMO	○	HMO	○	HMO	HHO, HMO	HMO	○	○	○	Play 1
○	○	○	○	○	○	○	○	○	○	HHO, HMC	○	○	○	○	Play 2
○	○	○	○	○	○	○	○	○	○	(HHO), (HMC)	○	○	○	○	Rails
○	○	○	○	○	○	○	○	○	○	HHO	○	○	○	○	Spring Boot
○	○	○	○	○	○	○	○	○	○	(HHO), (HMC)	○	○	○	○	Symfony

Table 5: CPDoS vulnerability overview

Top 500 websites. In addition to this, we used the Google Big Query service to investigate over 365 million URLs stored in the HTTP Archive data set `httparchive.summary_requests.2018_12_15_-desktop`. Table 6 shows the number of websites and URLs of the DoD, the Alexa Top 500 and the HTTP Archive where the response header indicates that the content is processed by a vulnerable HTTP implementation.

	DoD	Alexa Top 500	HTTP Archive
Total number of web sites/URLs	414	500	365.112.768
Varnish	2	40	4.658.950
Akamai	2	38	1.031.535
CDN77	0	0	321.456
Cloudflare	7	34	18.236.800
CloudFront	8	23	12.140.461
Fastly	0	9	4.013.578
IIS	27	9	17.792.692
Flask	0	0	5.765
Play 1	0	0	10.491

Table 6: Number of websites/URLs using Varnish, Akamai, CDN77, Cloudflare, CloudFront, Fastly, IIS, Flask and Play 1

The results highlight that eight websites of the DoD, 23 of the Alexa Top 500 and over twelve million URLs stored in the mentioned data set of the HTTP Archive are served via CloudFront. Moreover, all eight websites of the DoD, 16 websites of the Alexa Top 500 and over nine million URLs of the HTTP Archive point out that CloudFront in combination with Apache HTTPD, Nginx, Amazon S3, Microsoft IIS and Varnish is used. Our experiments revealed that these constellations are vulnerable to CPDoS attacks (see Table 5). However, it is very difficult to estimate the exact number of vulnerable websites without inspecting each of them individually. Moreover, the experiments have been done with the default configuration and without taking any other intermediate system into account. It is, however, very common that content providers change the default configuration of a cache in order to adapt the caching policy to the respective needs. Moreover, real world web applications also utilize other intermediate systems such as load balancers or WAFs. All these settings influence the practicability of CPDoS attacks in any direction. To get a clearer picture on the real life impact of CPDoS attacks, we took some samples based on the URLs from the Alexa Top 500, DoD, and HTTP Archive data sets. Overall, we found twelve vulnerable resources within a few days. These also include mission-critical websites such as ethereum.org, marines.com, and nasa.gov which use CloudFront as CDN. At all these websites, we were able to block multiple resources including scripts, style sheets, images, and even dynamic content such as the start page. The visual damage of a CPDoS attack is shown by the Figures 6 and 7 in the Appendix A. In Figure 6, the CPDoS attack is first applied to an image referenced in the start page of the victim website ethereum.org. Then the style sheet file is denied and finally, an error page replaces the whole start page. Figure 7 illustrates the affected start page of marines.com which displays an error page to the user instead of the genuine content. Moreover, we were also able to conduct a successful CPDoS attack on the update files of IKEA's Smart Home devices. IKEA uses CloudFront in conjunction with S3 to distribute remote control firmware and driver updates for their wireless bulbs. As CloudFront in combination with S3 is vulnerable to HHO CPDoS attacks, an attacker can block the remote control devices of IKEA from fetching security patches. These evidences show that CPDoS attacks can affect static as well as dynamic resources. Most of the vulnerable websites use CloudFront as CDN. However, the real world impact of CPDoS attacks is not only bound to CloudFront. We also found vulnerable websites in our sample which utilize other CDNs such as Akamai or Cloudflare in conjunction with Play 1. We have uncovered these examples in a few days only. An advanced attacker with political and financial motivation is easily able to gather much more vulnerable resources

as they only need to investigate the response headers in order to estimate whether a target website or resource is potentially vulnerable to CPDoS attacks. Moreover, the freely available HTTP Archive data sets via Google Big Query include millions of URLs which can be investigated by an attacker. For instance, HTTP Archive data set `httparchive.summary_requests.2018_12_15_desktop` contains over 9 millions URLs which we considered as highly vulnerable since the response headers of these resources indicate that CloudFront in conjunction with Apache HTTPD, Nginx, Amazon S3, Microsoft IIS, and Varnish is used. Among them are also many critical websites and resources including Amazon itself, the website dowjones.com, as well as Logitech which distributes firmware via CloudFront.

5.7 Practical Considerations

Caches are only vulnerable to CPDoS attacks if they store and reuse error pages. Web caching systems such as Stackpath, CDNSun, KeyCDN and G-Core labs cannot be affected by CPDoS attacks, since these CDNs do not cache error messages at all. This is also true for Apache HTTPD, Nginx and Squid when using them as an intermediate cache without involving any other vulnerable web caching systems.

As with other cache poisoning vulnerabilities, CPDoS attacks are only possible when a vulnerable web caching system does not contain a fresh copy of the to be attacked resource. That is, if a shared cache still maintains and reuses a stored fresh response for recurring requests, a malicious request is not able to poison the intermediary. The web caching system serves all requests to the target resource. None of the requests are forwarded to the origin server until the freshness lifetime is expired, so that no error page can be triggered. This means if a cache still owns a fresh response, an attacker has to wait until the cached content is stale. The most straightforward information to find out the expiration time is the `Expires` header which indicates the absolute expiration date. If the response does not contain an `Expires` header or the expiration time of this header is overridden by the `max-age` or `s-maxage` directive control directive, the attacker can make use of the `Age` header. The `Age` header declares the seconds of stay in the cache. The value of the `Age` header subtracted from the value of the `max-age` or `s-maxage` directive is the relative expiration time of the cached response. If the cached response is expired, the attacker's request must be the very first request so that it can reach the origin server to trigger an error page. To increase the likelihood for being the first request, we send automatized requests with a one second interval when the response is close to expire. With this technique we were able to successfully attack all twelve vulnerable websites of our spot check experiment. Sending regularly performed requests with one second distance of time is also a useful approach for cached responses which does contain any expiration time information, i.e., resources which are implicitly cached. Such responses usually do not contain any `max-age` or `s-maxage` directives and `Expire` headers. Here, the attacker needs to send automatized requests until one of the requests is forwarded to the origin server. Moreover, automatized requests with a one second interval are not considered as harmful even when they are sent over a long time, since health checks requests can also have the same interval. We tested this technique on several CDNs which also included WAFs and DDoS protections. Since we only used a single client to perform the attack, none of CDNs detected the malicious requests.

Many web applications configure the proxy cache or the CDN to serve the whole website. This means all resources including dynamic pages and static files are forwarded and processed by the cache. To exclude dynamic pages from being implicitly cached, content providers include `no-store` or `max-age=0` to the response header, so that each request must be forwarded to the origin server. If a vulnerable cache in conjunction with a vulnerable server-side HTTP implementation is used, these resources can be attacked without the need to wait and any automation of sending requests. One single malicious request is enough to paralyze the target resource, since each request is forwarded to the origin server. Vulnerable websites which configure the CDN to serve all resources are, e.g., marines.com, ethereum.org and nasa.gov.

There also many web applications which only configure the cache to store and reuses responses of certain URL paths such as for static files in the javascript or images directory. Other URL paths are accordingly not cached at all. Many content providers also maintain subdomains (e.g. static.example.org) or a specific domain for static files which are served via a cache. In these cases, only resources within the cached URL paths or the specific domain can be affected. To find out whether a distinct response traverses a cache, an attacker can inspect the response headers. For instance, the `Age` response header indicate that a cache is utilized. The main website of IKEA (ikea.com) does not use CloudFront or any other vulnerable HTTP implementations which indicates that this homepage is most likely not vulnerable to CPDoS attacks. However, IKEA uses a specific domain (fw.ota.homesmart.ikea.net) in conjunction with CloudFront to host the update files of their Internet of Things devices.

Another important limitation of CPDoS attacks is that the web caching systems except Fastly do only cache error pages for few minutes or seconds. Fastly stores and reuses the error page for one hour. If this time span is over, then the first benign request to the target resource is forwarded to origin server and refreshed again. Still, to extend the duration of CPDoS attacks, malicious clients can resend harmful requests in accordance to the fixed interval.

6 RESPONSIBLE DISCLOSURE

All discovered vulnerabilities have been reported to the HTTP implementation vendors and cache providers on February 19, 2019. We worked closely with these organizations to support them in eliminating the detected threats. We did not notify the website owners directly, but left it to the contacted entities to inform their customers.

Amazon Web Services (AWS). We reported this issue to the AWS-Security team. They confirmed the vulnerabilities on CloudFront. The AWS-Security team stopped caching error pages with the status code `400 Bad Request` by default. However, they took over three months to fix our CPDoS reportings. Unfortunately, the overall disclosure process was characterized by a one-way communication. We periodically asked for the current state, without getting much information back from the AWS-Security team. They never contacted us to keep us up to date with the current process. For

example, we only got noticed about the changed default caching policy by checking back the revision history of their respective documentation hosted in Github. Thus, we do not have much information on the noticeable amount of time required to resolve our reported CPDoS vulnerability, although having asked for it explicitly. We can only assume that this delay has to do with the large number of affected users they had to test after implementing according countermeasures. Moreover, Amazon suggests users to deploy an AWS WAF in front of the corresponding CloudFront instance. AWS WAF allows defining rules which drop malicious requests before they reach the origin server.

Microsoft. Microsoft was able to reproduce the reported issues and published an update to mitigate this vulnerability. They assigned this case to CVE-2019-0941 [27] which is published in June 2019.

Play 1. The developers of the Play 1 confirmed the reported issues and provided a security patch which limits the impact of the `X-HTTP-Method-Override` header [6]. The security patch is included in the versions 1.5.3 and 1.4.6. Older version are not maintained by this security patch. Web applications which use older versions of Play 1 therefore should update to the newest versions in order to mitigate CPDoS attacks.

Flask. We reported the HMO attack to the developer team of Flask multiple times. Unfortunately, we have not received any answer form them so far and hence we have to assume, that Flask-based web applications are still vulnerable to CPDoS.

7 DISCUSSION

Using malformed requests to damage web applications is a well-known threat. Request header size limits and blocking meta characters are therefore vital means of protection to avoid known cache poisoning attacks as well as other DoS attacks such as request header buffer flow [26] and ReDoS [38]. Also, many security guidelines such as the documentation of Apache HTTPD [2], OWASP [35], and the HTTP standard [12] recommend to block oversized headers and meta characters in headers. CPDoS attacks, however, aims to beat these security mechanisms with their own weapons. HHO and HMC CPDoS attacks intentionally send a request with an oversized header or harmful meta character with the intent to get blocked by an error page which will be cached. Along these lines, it is interesting to see that CDN services, which claim to be an effective measure to defeat DoS and especially DDoS attacks, desperately fail when it comes to CPDoS.

According to our experiment results, most of the presented attack vectors are only feasible when CloudFront is deployed as the underlying CDN, since it is the only analyzed cache which illicitly stores the error code `400 Bad Request`. Such a non-conformance is the main reason for the HHO and HMC attacks. The other major issue for both attacks is fact that the cache forwards oversized headers and requests with harmful meta characters. Violations of the HTTP standard and implementation issues are also the main reason for many other cache-related vulnerabilities including request smuggling, host of troubles, response splitting, and web deception attacks. The HMO CPDoS attack is, however, a vulnerability which does not exploit any implementation issues and violations of the HTTP standard. The `X-HTTP-Method-Override` header or similar headers are legitimate auxiliaries to tunnel HTTP methods which are not supported by WAFs or web browsers. Play 1 and Flask returns the error code `404 Not Found` or `405 Method Not Allowed` when an unsupported action in `X-HTTP-Method-Override` header is received. Both error messages are allowed to be cached according to RFC 7231. Akamai, CDN77, Fastly, Cloudflare, CloudFront, and Varnish follow this policy and cache such error codes. If these web caching systems are used in combination with one of the mentioned web frameworks, these combinations have an actual risk of falling victim to CDPoS attacks, even though they are in conformance with the HTTP standard and do not have any implementation issues. Therefore, the HMO CPDoS attack can be considered as a new kind of cache poisoning attack which does not exploit any implementation issues or RFC violations. This shows that CPDoS attacks do not always result from programming mistakes or unintentional violations of specification policies, but can also be the exploit of the conflict between two legitimate concepts. In case of HMO CPDoS attacks, this conflict refers to the usage of method overriding headers and the caching of allowed error messages.

Even though we did not detect attack vectors in other web caching systems and HTTP implementations, this does not mean that other constellations are not vulnerable to CPDoS attacks. As shown by Table 1 eight of fifteen tested web caching systems do store error pages and some of them even cache error pages which are not allowed. If an attacker is able to initiate other error pages or even cacheable error code at the target URL, then she may affect other web caching systems and HTTP implementations with CPDoS attacks as well. James Kettle, for instance, discovered two other forms of CPDoS attacks which fortunately are only successful due to specific implementation issues of the corresponding web application. The first CPDoS attack utilized the `X-Forwarded-Port` header [21]. This header usually informs the endpoint about the port that the client uses to connect to the intermediate system, which operates in front of the origin server. In the revealed attack, the cached response contained the redirect. A DoS was caused by the user's browser trying to follow the cached redirect and timing out. The second attack was able to create a DoS at www.tesla.com due to a faulty WAF configuration [20]. Tesla configured their WAF to block certain strings which have been used by other cache poison attacks. Unfortunately, requests with such strings were blocked by a `403 Forbidden` error page which was also cached. This shows that HMO, HHO, and HMC are not the only variations of CPDoS attacks. There are, certainly, many other ways to provoke an error page on the origin server. To the best of our knowledge and according to our experiences in developing web applications, it is not unlikely to provoke an `500 Internal Server Error` status code or other `5xx` errors in real world web applications and services. Akamai and Cloudflare do cache `5xx` error codes. At this point, we did not find a way to provoke such error messages in our experiments.

Moreover, we need to consider that contemporary web applications and distributed systems in particular are usually layered. That is, they often utilize other intermediate components such as load balancer, WAFs or other security gateways which are located between cache and endpoint. Such middleboxes or middleware may provide other request header size limits, meta character handling or header overriding features. Such systems may also react to malicious requests with error codes that could be cached.

8 COUNTERMEASURES

The most intuitive, as well as effective countermeasure, against CPDoS attacks is to exclude error pages from being cached. However, content providers which exclude cacheable error codes such as `404 Not Found` from being stored, need to consider that this setting may impair the performance and scalability. There two ways to exclude error pages from being cached. The first approach is to configure the web caching systems to omit the storage of error responses. Akamai, CDN77, CloudFront, CloudFront, Fastly, and Varnish provide options to do so. Content providers can also add the `no-store` directive to the `Cache-Control` response header which prohibits all caches from storing the content. According to our own evaluation, all tested web caching systems except CloudFront honored the keyword `no-store` in error pages and still do so. At the time of our experiments in February 2019, CloudFront cached error pages for five minutes by default and even did so when `no-store` was included in the error response header. The only way to avoid storing error pages in CloudFront was to disable each error code from caching via the CDN's configuration interface. Fortunately, AWS changed the behavior of caching error pages after our CPDoS reporting. One important change is that `400 Bad Request` error pages are not cached by default anymore. CloudFront only caches `400 Bad Request` error messages if they include a `max-age` or `s-maxage` control directive [1].

As mentioned before, the disobey of the HTTP standard in terms of ignoring control directives is the main cause for many cache-related vulnerabilities. Beside the consideration of cache-related control directives, web caching systems must, therefore, only store error codes which are permitted by the HTTP standard. Status codes such as `400 Bad Request` are not allowed to be cached, since this error message is only dedicated to a request which is malformed or invalid. Other error codes such as `404 Not Found`, `405 Method Not Allowed` or `410 Gone` can be cached, since they provide error information which is valid for all clients. Also, HTTP implementations have to use the appropriate status code for the corresponding error case. Table 3 shows that almost all tested system return the status code `400 Bad Request` for an oversized request header. IIS even replies with status the cacheable `404 Not Found` error code when a limit for a specific request header is exceeded. Both error messages are not the appropriate one for requests exceeding the header size limit. According to HTTP standard, the appropriate error code is `431 Request Header Fields Too Large`. Such error information is not stored and reused by any of the tested web caching systems. To test the compliance and behavior of caches, we recommend to use the cache testing tool of Nguyen et al. [31] or Mark Nottingham [33].

Another very effective countermeasure against CPDoS attacks is the usage of WAFs. Many CDNs provide the option to enable WAFs in order to protect web applications against malicious requests. To avoid CPDoS attacks, content providers can configure the WAF to explicitly block oversized requests, requests with meta characters or malicious headers. Using WAFs is, however, only effective if the WAF is implemented in the cache or in front of the cache, so that harmful requests can be eliminated before they are forwarded to the origin server. The experiments in Section 5 and the CPDoS attack of James Kettle on www.tesla.com [20] show that WAFs which are integrated at the origin server such as ModSecurity do not help against CPDoS attacks. Requests which are blocked by a WAFs at the origin can still trigger an error page that is stored by the cache.

Moreover, we recommend adding a subsection to the "Security Considerations" section of the RFC 7230 [12] to discuss the consequences of non-compliance with the protocol specification in order to avoid HHO, HMC and other web cache poisoning attacks. The "Security Considerations" section of RFC 7230 mentions cache-poisoning attacks including response splitting and request smuggling. However, the standard only makes recommendations that relate to these two specific attacks. The specification does not mention that the source of many cache-related attacks lies in violations of the standard. Such an additional description would increase developers' awareness of compliance with the specifications. HMO attacks, on the other hand, cannot be avoided by complying with the standard, as they are based on non-standard means which is the `X-HTTP-Method-Override` header in this case. To avoid HMO attacks while maintaining the scalability, content providers do not need to exclude the `404 Not Found` and `405 Method Not Allowed` error code from caching. Here, vulnerable web frameworks must follow the approach of Symfony, Lavarel as well as the plugins of Django and Express.js. These HTTP implementations support the method overriding headers, but only consider to change the action when the method in the request line is POST. By this, a `404 Not Found` error page cannot be triggered by malicious GET request, since method overriding headers are ignored. When trying to poison the cache with a POST request with a method override header including GET, the returning response is not stored by any tested cache. Also, the use of non-standard headers is a general approach to conduct other cache-poisoning attacks as described by James Kettle [22]. It is the responsibility of HTTP implementations to carefully integrate non-standard headers to avoid such attacks. To analyze impact of standardized or non-standard headers in respect to caches, developers and software testers can use, e.g., the testing tools of Nguyen et al. [31] and Mark Nottingham [33].

9 CONCLUSION AND OUTLOOK

Vulnerabilities stemming from the semantic gap result in serious security threats. Distributed systems are especially prone to such attacks as they are composed by distinct layers. Their existence is one major prerequisite for the different interpretation of an object, in this case the application messages floating through the intermediaries.

In this paper we extended the known vulnerabilities rooted in a semantic gap by introducing a class of new attacks, "Cache-Poisoned Denial-of-Service (CPDoS)". We systematically study how to provoke errors during request processing on an origin server and the case, in which error responses get stored and distributed by caching systems. We introduce three concrete CPDoS attack variations that are caused by the inconsistent treatment of the HTTP method override header, header size limits and the parsing of meta characters. We show the practical relevance by identifying the amount of available web caching systems that are vulnerable to CPDoS. The consequences can be severe as one simple request is sufficient to paralyze a victim website within a large geographical region (see Figure 8 in Appendix B). Depending on the resource

that is being blocked by an error page, the web page or web service can be disabled piecemeal (see Figure 6 in Appendix A).

According to our experiments 11% of the DoD web sites, 30% of the Alexa Top 500 websites and 16% of the URLs in the analyzed HTTP Archive data set are potentially vulnerable to CPDoS attacks. These cached contents include also mission-critical firmware and update files. Considering the fact that modern distributed applications often follow the Mircoservices [29] and Service-Oriented Architecture (SOA) [10] design principles where services are implemented with different programming languages and are operated by distinct entities, more semantic gap vulnerabilities may appear in the future. Hence, a more in-depth understanding of such vulnerabilities needs to be gathered in order to develop robust safeguards that do not depend on particular implementation and concatenation of system layers.

ACKNOWLEDGMENT

First of all, we would like to thank all reviewers for their thoughtful remarks and comments. Moreover, we would especially like to thank Shuo Chen and James Kettle for their feedback and suggestions. Finally, we appreciated the disclosure processes with the AWS-Security team, the Microsoft Security Response Center and the Play Framework development team.

This work has been funded by the German Federal Ministry of Education and Research within the funding program "Forschung an Fachhochschulen" (contract no. 13FH016IX6).

REFERENCES

[1] Amazon. 2019. How CloudFront Processes and Caches HTTP 4xx and 5xx Status Codes from Your Origin. https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/HTTPStatusCodes.html
[2] Apache HTTP Server Project. 2019. Security Tips. https://httpd.apache.org/docs/trunk/misc/security_tips.html
[3] G. Barish and K. Obraczke. 2000. World Wide Web caching: trends and techniques. *IEEE Communications Magazine* 38, 5 (2000), 178–184. https://doi.org/10.1109/35.841844
[4] M. Belshe, R. Peon, and M. Thomson. 2015. *Hypertext Transfer Protocol Version 2 (HTTP/2).* RFC 7540. IETF. https://tools.ietf.org/html/rfc7540
[5] T. Bray. 2016. *An HTTP Status Code to Report Legal Obstacles.* RFC 7725. IETF. https://tools.ietf.org/html/rfc7725
[6] A. Chatiron. 2019. Define allowed methods used in 'X-HTTP-Method-Override'. https://github.com/playframework/play1/issues/1300
[7] J. Chen, J. Jiang, H. Duan, N. Weaver, T. Wan, and V. Paxson. 2016. Host of Troubles: Multiple Host Ambiguities in HTTP Implementations. In *23th ACM SIGSAC Conference on Computer and Communications Security (CCS).* https://doi.org/10.1145/2976749.2978394
[8] G. Clemm and J. Whitehead J. Crawford, J. Reschke. 2010. *Binding Extensions to Web Distributed Authoring and Versioning (WebDAV).* RFC 5842. IETF. https://tools.ietf.org/html/rfc5842
[9] L. Dusseault. 2007. *HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV).* RFC 4918. IETF. https://tools.ietf.org/html/rfc4918
[10] T. Erl. 2007. *SOA Principles of Service Design.* Prentice Hall PTR.
[11] R. Fielding, M. Nottingham, and J. Reschke. 2014. *Hypertext Transfer Protocol (HTTP/1.1): Caching.* RFC 7234. IETF. https://tools.ietf.org/html/rfc7234
[12] R. Fielding and J. Reschke. 2014. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing.* RFC 7230. IETF. https://tools.ietf.org/html/rfc7230
[13] R. Fielding and J. Reschke. 2014. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content.* RFC 7231. IETF. https://tools.ietf.org/html/rfc7231
[14] Flask. 2010. Adding HTTP Method Overrides. http://flask.pocoo.org/docs/1.0/patterns/methodoverrides/
[15] O. Gil. 2017. WEB CACHE DECEPTION ATTACK. In *Blackhat USA.* https://blogs.akamai.com/2017/03/on-web-cache-deception-attacks.html
[16] K. Holtman and A. Mutz. 1998. *Transparent Content Negotiation in HTTP.* RFC 2295. IETF. https://tools.ietf.org/html/rfc2295
[17] IEEE Spectrum. 2018. Interactive: The Top Programming Languages 2018. https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018
[18] Suman Jana and Vitaly Shmatikov. 2012. Abusing File Processing in Malware Detectors for Fun and Profit. In *33rd IEEE Symposium on Security and Privacy.* 80–94. https://doi.org/10.1109/SP.2012.15
[19] Y. Jia, Y. Chen, X. Dong, P. Saxena, J. Mao, and Z. Liang. 2015. Man-in-the-browser-cache. *Computers and Security* 55, C (2015), 62–80. https://doi.org/10.1016/j.cose.2015.07.004
[20] J. Kettle. 2018. Bypassing Web Cache Poisoning Countermeasures. https://portswigger.net/blog/practical-web-cache-poisoning
[21] J. Kettle. 2018. Denial of service via cache poisoning . https://hackerone.com/reports/409370
[22] J. Kettle. 2018. Practical Web Cache Poisoning. In *Blackhat USA.* https://portswigger.net/blog/practical-web-cache-poisoning
[23] A. Klein. 2004. *Divide and Conquer - HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics.* White Paper. Sanctum, Inc. https://dl.packetstormsecurity.net/papers/general/whitepaper_httpresponse.pdf
[24] C. Linhart, A. Klein, R. Heled, and S. Orrin. 2005. HTTP REQUEST SMUGGLING. http://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf
[25] L. Masinter. 1998. *Hyper Text Coffee Pot Control Protocol (HTCPCP/1.0).* RFC 2324. IETF. https://tools.ietf.org/html/rfc2324
[26] NATIONAL VULNERABILITY DATABASE. 2010. *CVE-2010-2730 Detail.* CVE 2010-2730. Nist. https://nvd.nist.gov/vuln/detail/CVE-2010-2730
[27] NATIONAL VULNERABILITY DATABASE. 2019. *CVE-2019-0941 Detail.* CVE 2019-0941. Nist. https://nvd.nist.gov/vuln/detail/CVE-2019-0941
[28] Netcraft. 2019. January 2019 Web Server Survey. https://news.netcraft.com/archives/2019/01/24/january-2019-web-server-survey.html
[29] S. Newman. 2015. *Building microservices: designing fine-grained systems.* O'Reilly.
[30] H. V. Nguyen, L. Lo Iacono, and H. Federrath. 2018. Systematic Analysis of Web Browser Caches. In *2nd International Conference on Web Studies (WS).* https://doi.org/10.1145/3240431.3240443
[31] H. V. Nguyen, L. Lo Iacono, and H. Federrath. 2019. Mind the Cache: Large-Scale Analysis of Web Caching. In *34rd ACM/SIGAPP Symposium on Applied Computing (SAC).* https://doi.org/10.1145/3297280.3297526
[32] H. Nielsen and S. Lawrence. 2000. *An HTTP Extension Framework.* RFC 2774. IETF. https://tools.ietf.org/html/rfc2774
[33] M. Nottingham. 2019. HTTP Caching Tests. https://cache-tests.fyi/
[34] M. Nottingham and R. Fielding. 2012. *Additional HTTP Status Codes.* RFC 6585. IETF. https://tools.ietf.org/html/rfc6585
[35] OWASP. 2017. Denial of Service Cheat Sheet. https://www.owasp.org/index.php/Denial_of_Service_Cheat_Sheet#Mitigation_3:_Limit_length_and_size
[36] L. Richardson and S. Ruby. 2008. *RESTful web services.* O'Reilly Media, Inc.
[37] J. Somorovsky, M. Heiderich, M. Jensen, J. Schwenk, N. Gruschka, and L. Lo Iacono. 2011. All Your Clouds Are Belong to Us: Security Analysis of Cloud Management Interfaces. In *3rd ACM Workshop on Cloud Computing Security Workshop.* ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/2046660.2046664 http://doi.acm.org/10.1145/2046660.2046664.
[38] C.-A. Staicu and M.l Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in Javascript-based Web Servers. In *27th USENIX Conference on Security Symposium (USENIX Security).* USENIX Association, Berkeley, CA, USA, 361–376. http://dl.acm.org/citation.cfm?id=3277203.3277231
[39] S. Triukosea, Z. Al-Qudad, and M. Rabinovich. 2009. Content Delivery Networks: Protection or Threat?. In *14th European Symposium on Research in Computer Security (ESORICS).* https://doi.org/10.1007/978-3-642-04444-1_23

APPENDIX A: ILLUSTRATIVE EXAMPLES OF CPDOS ATTACK

A.1 Ethereum-website

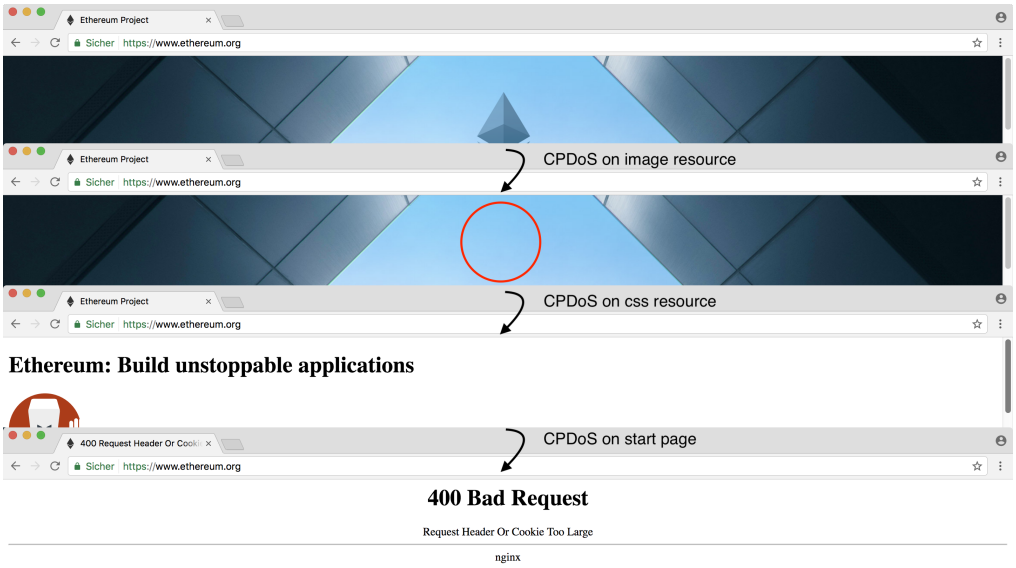

Figure 6: These screenshots show the start page of the website ethereum.org and how parts as well as the whole page are rendered inaccessible due to a successful CPDoS attack. More specifically, this website has been vulnerable to HHO CPDoS.

A.2 Marines-website

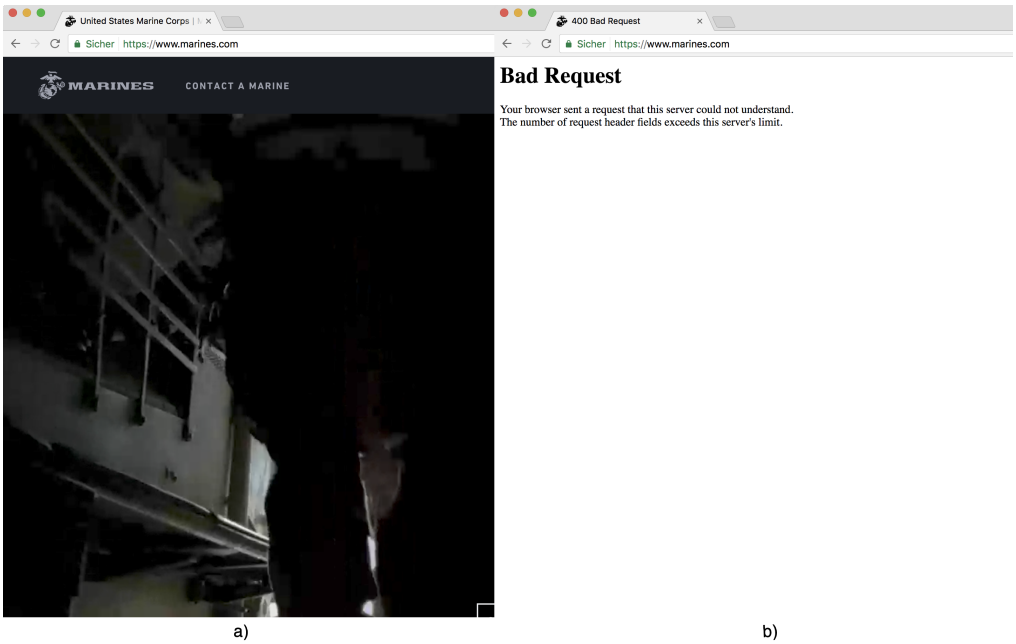

Figure 7: These two screenshots show the start page of the website marines.com before a) and after b) a successful CPDoS attack. More specifically, this website has been vulnerable to HHO CPDoS.

APPENDIX B: CPDOS ATTACK SPREAD

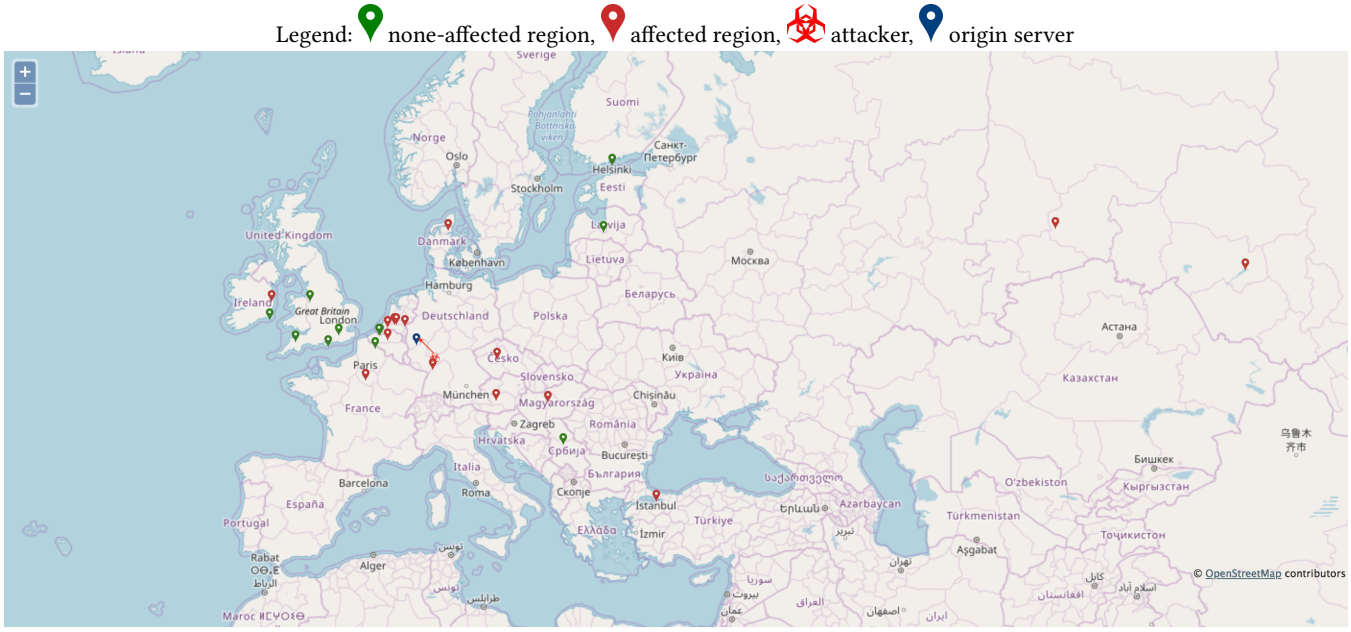

Figure 8: Affected CDN regions when sending a CPDoS attack from a) Frankfurt, Germany and b) Northern Virginia, USA to a victim origin server in Cologne, Germany.