Host of Troubles: Multiple Host Ambiguities in HTTP Implementations

Jianjun Chen[*†]
chenjj13@mails.tsinghua.edu.cn

Jian Jiang[‡]
jiangjian@berkeley.edu

Haixin Duan[*†]
duanhx@tsinghua.edu.cn

Nicholas Weaver[‡§]
nweaver@icsi.berkeley.edu

Tao Wan[¶]
tao.wan@huawei.com

Vern Paxson[‡§]
vern@berkeley.edu

[*]Tsinghua University, [†]Tsingua National Laboratory for Information Science and Technology
[‡]UC Berkeley, [§]ICSI, [¶]Huawei Canada

ABSTRACT

The Host header is a security-critical component in an HTTP request, as it is used as the basis for enforcing security and caching policies. While the current specification is generally clear on how host-related protocol fields should be parsed and interpreted, we find that the implementations are problematic. We tested a variety of widely deployed HTTP implementations and discover a wide range of non-compliant and inconsistent host processing behaviours. The particular problem is that when facing a carefully crafted HTTP request with ambiguous host fields (e.g., with multiple Host headers), two different HTTP implementations often accept and understand it differently when operating on the same request in sequence. We show a number of techniques to induce inconsistent interpretations of host between HTTP implementations and how the inconsistency leads to severe attacks such as HTTP cache poisoning and security policy bypass. The prevalence of the problem highlights the potential negative impact of gaps between the specifications and implementations of Internet protocols.

1. INTRODUCTION

Postel's law, also called the robustness principle, is commonly phrased as "Be conservative in what you do, be liberal in what you accept from others" [19]. Although this maxim is regarded as a good design principle for robust network systems, it may prove disastrous in an adversarial context. Attackers can exploit this permissiveness when two different devices interpret the same liberal response differently.

Perhaps the most permissive widely deployed protocol is HTTP. Although the request format is tightly specified [6], many implementations are quite broad in what they actually accept. Some variations appear harmless in a single product, but inconsistent interpretation between different parties can have drastic consequences.

The problem arises when an attacker can generate a direct HTTP request (such as by using Flash on a victim's web browser) where the request contains multiple, ambiguous mechanisms to define the target host, such as multiple Host headers or a Host header combined with an absolute-URI in the request-line. If one in-path device (such as a cache proxy or firewall) interprets the request one way but the final destination (such as a Content Delivery Network (CDN) or other co-hosting service providers) interprets it differently, the result may be an exploitable semantic inconsistency. These can enable cache poisoning and filter bypass, which we frame as reflecting a "Host of Troubles".

We conduct an in-depth empirical study to understand how inconsistent interpretation of Host can manifest between different HTTP implementations and what kind of security consequence it leads to. We find significantly different behaviour among 33 popular HTTP implementations, leading to three exploiting techniques: (a) multiple Host headers, (b) space-surrounded Host headers, and (c) requests with absolute-URI. We identify a large number of combinations of downstream and upstream HTTP implementations where an inconsistent host semantic could occur. We find that such semantic inconsistency can lead to severe security consequences such as cache poisoning and filtering bypass.

Overall, we make three main contributions:

1) We present a class of new attacks, "Host of Troubles", that broadly threaten the Internet. Our study shows that these attacks affect numerous HTTP implementations, sometimes severely. One of the exploits we found in Squid allows an attacker to remotely poison the cache of *any* HTTP website with arbitrary content.

2) We systematically study the behavior of 33 HTTP implementations in their handling of `Host` headers and identify a large range of interpretation inconsistencies that attackers can leverage for cache poisoning and filtering bypass. We have reported these to CERT/CC and affected vendors, who are actively addressing them.

3) We conduct a large scale measurement of transparent caches on the Internet and discover that around *97% of users* served by a transparent cache are subject to cache poisoning attacks we found. We provide an online checker for users to evaluate whether their networks are vulnerable to such attacks.

2. BACKGROUND

An HTTP request consists of a request start-line, zero or more request headers and an optional message body. The

CCS'16, October 24 - 28, 2016, Vienna, Austria

ISBN 978-1-4503-4139-4/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2976749.2978394

request-line and the request headers specify the HTTP protocol fields. One of the most important purposes of the protocol fields in a request is for recipient to locate the requested resource. In HTTP/1.0, the only field for this purpose is a request-target in the request-line. The request-target can be a host-relative path starting with "/", or an absolute-URI composed of a schema, a host, and a path. The latter is designed to support resource access through a proxy, although end systems will also accept absolute-URIs in the request-line. HTTP/1.1 introduced a Host header to support request routing in a co-hosting environment where multiple websites deploy on a same IP address. These websites are isolated by different domains.

HTTP is a client-server protocol with explicit support of intermediates. Five categories of intermediates are commonly deployed and relevant in this study: forward proxies, interception proxies (transparent caches), reverse proxies, Content Delivery Networks (CDNs), and firewalls. For purposes of this paper, we will designate the first device as "downstream" which can forward the request to the "upstream" device [1].

A forward proxy is explicitly configured in a client, such as a web browser, to handle all web requests for that client. Forward proxies are commonly configured for either performance, filtering, privacy, mandatory censorship, or censorship-evasion reasons. It is also possible to chain forward proxies, where one forward proxy passes all requests to another forward proxy.

An interception proxy requires neither client nor server interaction. To intercept all web requests, an interception proxy usually depends on a network device that uses policy based routing to forward all web packets (i.e., with source or destination TCP port of 80 or 443) to the proxy. The interception proxy then inspects all web requests before forwarding them onward. The most common use for such proxy is content caching [2]. Therefore it is also commonly referred as a transparent cache. Internet Services Providers (ISPs) deploy transparent caches to improve performance and to localize network traffic [30].

A reverse proxy is deployed in front of one or more servers. Clients directly issue requests to the reverse proxy, which then retrieves resources from servers. Reverse proxies are usually configured to provide features that are independent of web applications. Common features include load balancing, caching, TLS/SSL termination, and content filtering.

A CDN is essentially a reverse proxy service provided by a third party. A CDN can provide desired reverse proxy features with a large number of nodes that are geographically close to end-users. Once authorized by a website (usually by configuring the site's DNS), requests to the website are directly sent to near-by nodes of the CDN provider. These CDN nodes either serve the requests with cached content or forward to original server of the website. CDNs are widely adopted as they offer substantial benefits to both latency and available bandwidth.

Firewalls are commonly deployed on end-hosts or at network edge, inspecting passing-through traffic to enforce various security policies. The security policy relevant to this work is website blacklisting, in which a firewall examines HTTP requests to block access to unwanted websites (e.g., by injecting TCP resets or dropping packets).

An important and relevant difference between these intermediates is how they handle HTTPS traffic. Forward proxies, transparent caches, and network-based firewalls are not capable of inspecting HTTP messages over HTTPS connections unless they act as TLS/SSL man-in-the-middle. In comparison, HTTPS connection can always terminate at an HTTPS capable reverse proxy or CDN node.

[1]RFC 7230 [6] has an inconsistent definition, as the roles of "upstream" and "downstream" are switched whether it is a request or a response, since it just specifies that all messages flow from upstream to downstream.

[2]The other primary use is mandatory censorship in corporate networks.

3. MULTIPLE HOST AMBIGUITIES

Generally, processing an HTTP request can be divided into two phases: in the first phase, the textual message is firstly parsed to *recognize* valid protocol fields, and the recognized protocol fields are interpreted into a semantic structure; in the second phase, the semantic structure is then used for further actions. A request with invalid protocol fields should be *rejected* in the first phase with Client Error 4XX responses.

In parsing and interpreting the HTTP semantics, one of the most important designations is what host is involved with the request, because Host is the key protocol field for resource locating, request routing, caching, etc. The problem of multiple host ambiguities arises when two parties in an HTTP processing-chain parse and interpret host in a crafted, adversarial request differently. Inconsistency of host between two parties often causes disastrous consequences because of its semantic importance.

Two parties of one HTTP processing-chain can be connected either in parallel or in series. For the former case, two parties receive same request simultaneously. Discrepancies in parsing and interpreting between the two directly result in a semantic inconsistency of host. For example, an Intrusion Detection System (IDS) and its protected server are usually connected in parallel, Inconsistency of host between them may enable IDS evasion if the IDS is looking for a particular host. In the latter case, a downstream party receives and processes a request, then *forwards* it to an upstream device. In such case, different parsing and interpreting behaviours are not sufficient to cause semantic inconsistency between the downstream and the upstream. How the downstream forwards the request also plays a necessary role in whether semantic inconsistency could occur. Inconsistent interpretation of host can be avoided if the downstream always forwards a *normalized* request that is unambiguous with its own interpretation.

We assess the problem of multiple host ambiguities in deployed HTTP systems by conducting black-box testing on a total of 33 widely used HTTP implementations, including 6 servers, 2 transparent caches, 3 forward proxies, 7 reverse proxies, 8 CDNs, and 7 firewalls. Table 1 presents the names and versions of the tested implementations. Some programs support multiple configurations. For these programs, we test their typical working modes and count them as different implementations in corresponding categories. For example, Squid can be configured as three modes: transparent cache, forward proxy, and reverse proxy. We test it in all three modes respectively, and would therefore count this as 3 tested implementations. Hereinafter, we use "name (category)" to refer specific tested implementations.

Prior experience of HTTP specifications and implementa-

Category	Implementation (version)
Server	Apache (2.4.20), IIS (8.5), Lighttpd (1.4.39), LiteSpeed (5.0.16), Nginx (1.9.13), Tomcat (8.0.33)
Transparent Cache	Apache Traffic Server ATS (6.1.1), Squid (3.5.16)
Forward Proxy	Apache, IIS, Squid
Reverse Proxy	Apache, IIS, Lighttpd, LiteSpeed, Nginx, Squid, Varnish (4.1.2)
CDN	Akamai, Alibaba, Azure, CloudFlare, CloudFront, Fastly, Level3, Tencent
Firewall	Bitdefender (Internet Security 2016 on Win8.1), ESET (Cyber Security Pro 6.1.12.0 on Mac), Huawei (USG 6370 Next Generation Firewall), Kaspersky (Internet Security 2016 on Win8.1), OS X (El Capitan 10.11.4), Palo Alto Networks PAN (PA-7050), Windows (8.1 Pro)

Table 1: Tested HTTP implementations.

tions lead us to develop test cases based on three techniques: multiple Host headers, a space-surrounded Host header, and using an absolute-URI as a request-target. We first measured how the implementations parse and interpret crafted requests with various ambiguous host in which we find a large number of differences (Table 2 and Table 3). We then turn to resolve their forwarding behaviours, because in practice the implementations we test are typically connected in series. We found that 21 out of 33 implementations do not normalize requests sufficiently when forwarding them to upstream (Table 4). With these knowledge, we further examine 396 selected downstream-upstream pairs of the tested implementations. In total we identify 202 cases where a specially crafted request can cause inconsistent understanding of which host the request should be attributed to between the downstream and the upstream systems (Table 5). In addition, we also discovered one case where different host interpretation occurs between different internal modules in one implementation.

In the rest of this section, we first explain how we explore the three testing techniques, illustrated with representative cases. We then present some further details of our findings.

3.1 Multiple Host Headers

Specifications. RFC 2616 [5] states that a request with multiple same name headers is allowed only if the value of this header is defined as a single comma-separated list, which implies that a request with multiple Host headers is invalid. RFC 7230 [6] explicitly specifies that requests with multiple Host headers must be reject with 400 Bad Request.

Implementations. We find that 25 out of 33 tested implementations do not follow the specifications to reject requests containing multiple Host headers. Apache (Server, Reverse Proxy) concatenate multiple Host headers with a comma (implicitly combining the multiple headers into a single invalid host), and OS X (Firewall) likely behaves in the same way. Among the rest 22 implementations, all take the first header except Tencent (CDN) and ESET (Firewall), which take the last header.

Inconsistent interpretation of the hostname happens between an upstream and a downstream if they have different preference of multiple Host headers, and the downstream forwards multiple, ambiguous Host headers to the upstream system. Figure 1(a) shows such an example. However, we find that in many cases, the downstream performs some form of normalization such that the forwarded request does not contain multiple and different Host headers. For example, the same technique in Figure 1(a) does not work when Squid (Transparent Cache) as downstream and Tencent (CDN) as upstream, because Squid (Transparent Cache) changes all recognized Host headers to the value it interprets. Interestingly, these normalizations are often insufficient when spaces come into play.

3.2 Space-surrounded Host Header

Specifications. Space around a header name can appear in three forms: the first header with preceding spaces, other headers with preceding spaces, and headers with succeeding spaces. RFC 2616 does not have explicit text for the first and the third case. The syntax definition implies that systems should reject the former and allow the latter. For the second case, RFC 2616 states that a such header needs to be processed as folded line of its previous header: remove its preceding line break characters to concatenate with the previous header.

RFC 7230 has explicit text description for each case. For the first, it suggests to either reject the request or ignore the header. For the second case, although RFC 7230 already obsoletes line folding, it still allows a proxy or a server to process as line folding for backward compatibility considerations. The third case is explicitly forbidden.

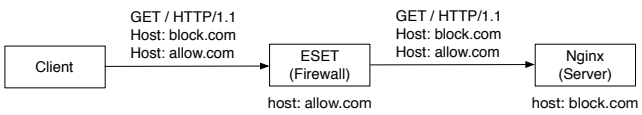
Implementations. We find that implementations vary largely in processing space-surrounded Host headers. Table 2 presents detailed behaviours for each implementation. Notably, we observe 10 distinct behaviours among 33 implementations. Only 5 implementations comply with RFC 2616 and 2 comply with RFC 7230. In addition, when acting as upstream, 16 implementations appear to forward space-surrounded Host headers to the upstream under certain conditions (see Table 4 for details).

These behaviours open new opportunities of multiple host ambiguities. In most cases, different understanding of host happens between an upstream and a downstream because they interprets space-surrounded Host headers differently. For example, in Figure 1(b), Squid (Transparent Cache) sees the space-preceding Host header as an unknown header. Therefore it forwards the space-preceding Host header without normalization. However, the downstream, Tencent (CDN), recognizes the space-preceding Host header as valid Host header and accepts its value as interpreted host because it prefers the last of multiple Host headers.

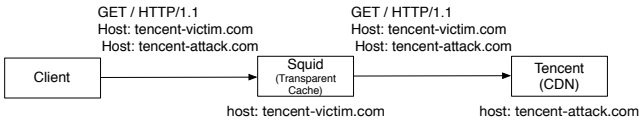
Sometimes, even the upstream and the downstream have similar or same host parsing and interpreting logic, they may still be fooled to interpret one request differently because of special forwarding behaviours of the upstream. Figure 1(c) shows such an example. Both Akamai (CDN) and Squid (Reverse Proxy) prefer the first of multiple Host headers, they also have similar behaviours in processing space-preceding Host header. However, in certain cases, Akamai (CDN) "flips" space-preceding Host header and normal Host header when forwarding a request. The flipped request causes a downstream Squid (Reverse Proxy) to interpret a different host.

3.3 Absolute-URI as Request-Target

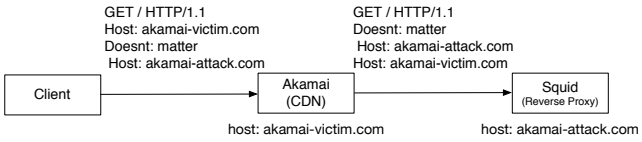
As we explained in Section 2, HTTP allows a client to send absolute-URI as request-target, which contains a host



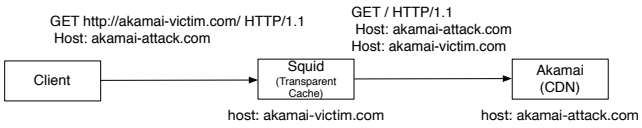
(a) Preference of multiple Host headers.



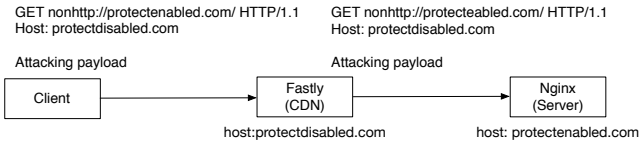
(b) Multiple Host headers combined with preceding space.



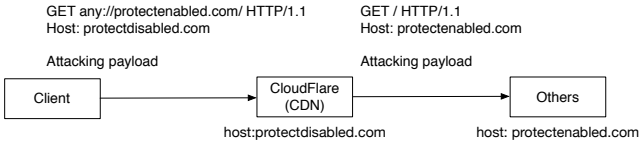
(c) Exploiting "flipped" forwarding.



(d) Absolute-URI with Host header.



(e) Schema of absolute-URI.



(f) Interpreting absolute-URI, forwarding Host header.

Figure 1: Different cases of inconsistent interpretation of host between upstream and downstream.

component. It turns out the intervention between host component in absolute-URI and Host header is another vector for multiple host ambiguities.

Specifications. Both RFC 2616 an RFC 7230 require server to accept absolute-URI as request-target, and to prefer host component of absolute-URI than Host header. RFC 7230 additionally requires requests with absolute-URI to have identical host component as Host header. Both of the two RFCs do not explicitly state which schema is allowed in the absolute-URI.

Implementations. We find that implementations vary in recognizable schema of absolute-URI. While some recognize host in absolute-URI with any schema, some only support HTTP and/or HTTPS schemas, ignoring or rejecting absolute-URI with unsupported schemas. Few implementations do not recognize a hostname in an absolute-URI. For implementations recognizing a hostname in absolute-URI, all except Akamai (CDN) comply with RFCs to prefer the host in the absolute-URI over Host header. But only Azure (CDN) enforces the identicality check required by RFC 7230. When forwarding, most implementations rewrite the absolute-URI to its path and add a Host header, except that LiteSpeed (Reverse Proxy) forwards absolute-URI to upstream unconditionally. Lighttpd (Reverse Proxy), Varnish (Reverse Proxy), and Fastly (CDN) also forward absolute-URIs when they do not recognize the schema.

In general, absolute-URIs enables two kinds of host ambiguities between an upstream and a downstream. First, when the downstream recognizes host in an absolute-URI, and rewrites it to path before forwarding, the upstream may recognize a (space-surrounded) Host header that is different from the host in the absolute-URI. For example, as shown in Figure 1(d), a Squid (Transparent Cache) takes the host from absolute-URI. However, the upstream, Akamai (CDN), interprets the host of the forwarded request using the different, space-preceding Host header. Second, if the downstream forwards the absolute-URI as-is, then different interpretation of the absolute-URI between the upstream and the downstream may cause inconsistent interpretation of host. In Figure 1(e), Fastly (CDN) does not recognize host in the absolute-URI because the schema is not HTTP. It takes Host header, and forwards the absolute-URI to the upstream. Instead, the upstream Nginx (Server), which recognizes the host in an absolute-URI with any schema, will interpret the forwarded request with the host in the absolute-URI.

We also found a case where absolute-URI causes inconsistent host between internal modules of one implementation. We present the details in Section 4.1.

3.4 Upstream-Downstream Combinations

We examine upstream-downstream combinations that we believe have some real-world deployment. Generally the downstream can be a transparent cache, a forward proxy, a reverse proxy, a CDN, or a firewall while the upstream can be another reverse proxy, CDN, or server. Among these combinations, we exclude the cases where the downstream is a reverse proxy and the upstream is a CDN because we are not aware of real-world case of such a scenario. We also exclude self-chaining of CDNs because these cases are considered harmful and CDNs should reject these [2].

128 out of 202 cases of host inconsistency are between firewalls (downstream) and other implementations (upstream). The main reason is that all tested firewalls but Bitdefender do not modify requests when forwarding. For each firewall, its parsing and interpreting behaviours are sufficiently different from most of other implementations so that we can find ways to cause a different interpretation of host. The only exception, Bitdefender, likely fails open when processing a request with absolute-URI.

CloudFlare has a unique forwarding behaviour that always and only forwards the first Host header. Because CloudFlare recognizes the host component in an absolute-URI with any schema, a request presented in Figure 1(f) is sufficient to cause host inconsistency between CloudFlare and any possible upstream.

Implementation/ Specification		Space-preceded Host as first header	Other space-preceded Host header	Space-succeeded Host header	schema of absolute-URI
Server	Apache	Not recognize	Line folding	Recognize	Recognize HTTP, not others
	IIS	Recognize	Line folding	Recognize	Recognize HTTP/S, reject others
	Lighttpd	Reject	Line folding	Recognize	Recognize HTTP/S, not others
	LiteSpeed	Reject	Line folding	Recognize	Recognize any schema
	Nginx	Not recognize	Not recognize	Not recognize	Recognize any schema
	Tomcat	Not recognize	Line folding	Not recognize	Recognize HTTP/S, reject others
Transparent Cache	ATS	Not recognize	Not recognize	Not recognize	Recognize any
	Squid	If no host before: recognize, else: not recognize	If no host before: recognize, else: not recognize	If no host before: reject, else: recognize	Recognize HTTP, reject others
Forward Proxy	Apache	Not recognize	Line folding	Recognize	Recognize HTTP, reject others
	IIS	Recognize	Line folding	Recognize	Recognize HTTP/S, reject others
	Squid	If no host before: recognize, else: not recognize	If no host before: recognize, else: not recognize	If no host before: reject, else: recognize	Recognize HTTP, reject others
Reverse Proxy	Apache	Not recognize	Line folding	Recognize	Recognize HTTP, not others
	IIS	Recognize	Line folding	Recognize	Recognize HTTP/S, reject others
	Lighttpd	Reject	Line folding	Recognize	Recognize HTTP/S, not others
	LiteSpeed	Reject	Line folding	Recognize	Recognize any schema
	Nginx	Not recognize	Not recognize	Not recognize	Recognize any schema
	Squid	If no host before: recognize, else: not recognize	If no host before: recognize, else: not recognize	If no host before: reject, else: recognize	Recognize HTTP, reject others
	Varnish	Reject	Line folding	Reject	Recognize HTTP, not others
CDN	Akamai	If no host before: recognize, else: not recognize	If no host before: recognize, else: not recognize	Reject	Recognize HTTP/S, reject others
	Alibaba	Not recognize	Not recognize	Not recognize	Recognize any schema
	Azure	Reject	Line folding	Recognize	Recognize HTTP/S, reject others
	CloudFlare	Not recognize	Not recognize	Not recognize	Recognize any schema
	CloudFront	Not recognize	Not recognize	Not recognize	Recognize any schema
	Fastly	Reject	Line folding	Reject	Not recognize any schema
	Level3	Not recognize	Not recognize	Reject	Recognize HTTP/S, reject others
	Tencent	Recognize	Recognize	Recognize	Recognize HTTP, reject others
Firewall	Bitdefender	Recognize	Recognize	Recognize	Likely fail-open
	ESET	Not recognize	Not recognize	Not recognize	Recognize any schema
	Huawei	Not recognize	Not recognize	Not recognize	Recognize any schema
	Kaspersky	Not recognize	Not recognize	Not recognize	Recognize any schema
	OS X	Not recognize	Not recognize	Not recognize	Not recognize any schema
	PAN	Not recognize	Not recognize	Not recognize	Recognize HTTP/S, not others
	Windows	Recognize	Recognize	Recognize	Recognize any
Specification	RFC 2616	Reject (implicit)	Line folding	Recognize	Not specified
	RFC 7230	Reject or not recognize	Reject or line folding	Reject	Not specified

Table 2: Host parsing behaviours: specifications and tested implementations ("recognize" means accepting as valid host field, "not recognize" means either ignoring or accepting as an unknown header field, "reject" means responding with 400 Bad Request).

4. EXPLOITATIONS

The presence of ambiguous chains enables potential exploitation. We have observed two types of exploitations: cache poisoning and filtering bypass. Each exploitation has two different forms.

4.1 HTTP Cache Poisoning

The first form of cache poisoning exploits the inconsistency between internal modules of Squid (Transparent Cache) to attack *any unencrypted website*. Therefore we call it *general cache poisoning*. The scenario requires an attacker who can send HTTP requests that pass through a shared transparent cache (Squid); "attack.com" controlled by the attacker and "victim.com" as the victim site, illustrated in Figure 2.

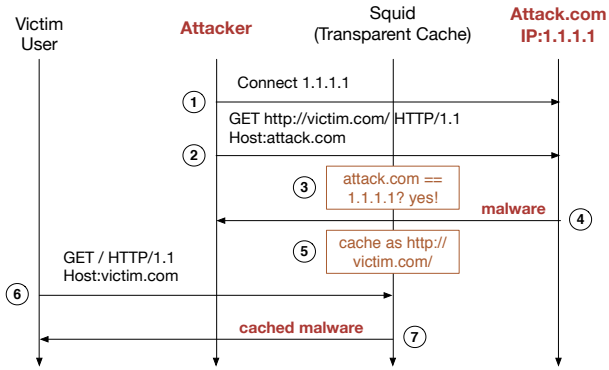


Figure 2: General cache poisoning of any unencrypted website on a Squid transparent cache.

The attacker first establishes a TCP connection to the HTTP server at "attack.com". Since the Squid proxy operates in a transparent fashion, it intercepts and mediates this connection. The attacker then issues an HTTP request with "victim.com" in absolute-URI and "attack.com" as Host header over this connection. Squid identifies the request as going to "victim.com". When it inspects the destination IP address for consistency, however, it mistakenly checks it against the value of the Host header, "attack.com", rather than "victim.com". Thus, the proxy directly passes the request to the "attack.com" server, but caches the (malicious) reply the server returns as a resource of "victim.com".

The second form of cache poisoning exploits the inconsistency between a downstream and an upstream, poisoning cache on the downstream to attack websites hosting on the upstream. We call it *co-hosting cache poisoning* because this attack needs a co-hosting upstream that provides access for both victim website and a website under attacker's control.

Implementation /Specification		Multiple Host headers	Presence of host			Recognized absolute-URI vs. Recognized Host header	
			Host header	Absolute-URI	Absent	Preference	Consistency
Server	Apache	Concatenate	Must	Optional	Reject	Absolute-URI	Optional
	IIS	Reject	Must	Optional	Reject	Absolute-URI	Optional
	Lighttpd	Reject	Optional	Optional	Reject	Absolute-URI	Optional
	LiteSpeed	Prefer first	Optional	Optional	Allow	Absolute-URI	Optional
	Nginx	Prefer first	Must	Optional	Reject	Absolute-URI	Optional
	Tomcat	Prefer first	Optional	Optional	Reject	Absolute-URI	Optional
Transparent Cache	ATS	Prefer first	Optional	Optional	Reject	Absolute-URI	Optional
	Squid	Prefer first	Optional	Optional	Allow	Absolute-URI	Optional
Forward Proxy	Apache	Use absolute-URI	Must	Must	Reject	Absolute-URI	Optional
	IIS	Reject	Must	Optional	Reject	Absolute-URI	Optional
	Squid	Use absolute-URI	Optional	Must	Reject	Absolute-URI	Optional
Reverse Proxy	Apache	Concatenate	Must	Optional	Reject	Absolute-URI	Optional
	IIS	Reject	Must	Optional	Reject	Absolute-URI	Optional
	Lighttpd	Reject	Optional	Optional	Reject	Absolute-URI	Optional
	LiteSpeed	Prefer first	Optional	Optional	Allow	Absolute-URI	Optional
	Nginx	Prefer first	Must	Optional	Reject	Absolute-URI	Optional
	Squid	Prefer first	Optional	Optional	Allow	Absolute-URI	Optional
	Varnish	Reject	Optional	Optional	Allow	Absolute-URI	Optional
CDN	Akamai	Prefer first	Optional	Optional	Reject	Host header	Optional
	Alibaba	Prefer first	Must	Optional	Reject	Absolute-URI	Optional
	Azure	Reject	Must	Optional	Reject	Absolute-URI	Must
	CloudFlare	Prefer first	Must	Optional	Reject	Absolute-URI	Optional
	CloudFront	Prefer first	Must	Optional	Reject	Absolute-URI	Optional
	Fastly	Reject	Must	—	Reject	—	—
	Level3	Prefer first	Optional	Optional	Reject	Absolutea-URI	Optional
	Tencent	Prefer last	Must	Optional	Reject	Absolute-URI	Optional
Firewall	Bitdefender	Prefer First	Optional	Optional	Allow	Likely fail-open	Optional
	ESET	Prefer last	Optional	Optional	Allow	Absolute-URI	Optional
	Huawei	Prefer first	Optional	Optional	Allow	Absolute-URI	Optional
	Kaspersky	Prefer first	Optional	Optional	Allow	Absolute-URI	Optional
	OS X	Likely concatenate	Optional	—	Allow	—	—
	PAN	Prefer first	Optional	Optional	Allow	Absolute-URI	Optional
	Windows	Prefer first	Optional	Optional	Allow	Absolute-URI	Optional
Specification	RFC 2616	Reject (implicit)	Must	Forward proxy: must Others: optional	Reject	Absolute-URI	Not specified
	RFC 7230	Reject	Must	Forward proxy: must Others: optional	Reject	Absolute-URI	Must

Table 3: Host interpreting behaviours: specifications and tested implementations.

Figure 1(d) provides an example where an attacker signs up with Akamai using "akamai-attack.com" to attack another Akamai customer "akamai-victim.com". The attacker issues a malicious request, fooling Squid to interpret the request as belonging to "akamai-victim.com", yet Akamai understands this as going to "akamai-attack.com" and forwards to a server under attacker's control. Consequently, the Squid caches a malicious response returned by "akamai-attack.com" as a resource of "akamai-victim.com". We confirm that ATS, Apache, Squid, Akamai, Alibaba, CloudFront are affected when acting as downstream with caching and chaining with a co-hosting upstream. Lighttpd, Varnish, CloudFlare, and Fastly are not affected because the exploiting requests interfere with their caching mechanisms.

Both forms of cache poisoning are remotely exploitable. Attackers can readily obtain the necessary vantage point using techniques such as Flash ads.

4.2 Filtering Bypass

The other significant attack vector is filtering bypass, where a downstream detects and filters "unwanted" HTTP requests not to reach an upstream, yet requests that exploit host inconsistency between the upstream and the downstream evade the downstream's filtering.

The first form of filtering bypass affects a firewall's website blacklisting. In Figure 1(a), ESET blacklists "block.com". Yet when a client connects to the server of "block.com", and issues a crafted request, ESET is fooled to believe the request is going to "allow.com" which is not blacklisted. When the request reaches the server, it identifies as "block.com" and returns content that suppose to be blocked.

The other form of filtering bypass evades protections provided by co-hosting upstreams, such as some security features of a CDN. Figure 1(f) and Figure 1(e) show how such attack could happen on websites hosted on CloudFlare and Fastly. An attacker signs up with CloudFlare or Fastly with "protectdisabled.com" to attack "protectenabled.com", which is protected by security features of CDN. The attacker first disables all security protection of "protectdisabled.com", and configures the forwarding destination as the original IP of "protectenabled.com". Then the attacker sends a malicious request with an ambiguous host and attacking payload (e.g., to exploit SQL injection). The ambiguous host causes CloudFlare or Fastly to believe that the request belongs to "protectdisabled.com" therefore it does not enforce any security policy. However, the upstream identifies the requests as going to "protectenabled.com" and sees it is forwarded by IPs of its CDN providers. Therefore the upstream trusts the request as benign and serves without further checks. This attack requires the attacker to uncover the target website's original IP that is supposed to be hidden. Previous research shows this pre-condition is possible in many cases due to imperfect operations [28] or simple mass scanning [4].

5. MEASURING TRANSPARENT CACHES

Among all potential exploitations we have found, we suggest that the poisoning of transparent caches is of most con-

Implementation		Simplified Description
Transparent Cache	ATS	1. for absolute-URI, rewrite to path; use its host to change the first recognized Host header, or add a new Host header before original headers; 2. forward all (other) recognized Host headers as-is; 3. forward space-preceded Host headers and space-succeeded Host headers as-is under certain conditions.
	Squid	1. for absolute-URI, rewrite to path; 2. change all recognized Host headers (except space-preceded ones) to the interpreted host, or add a new Host header after original headers; 3. forward space-preceded Host headers as-is.
Forward Proxy	Apache	1. rewrite absolute-URI to path, use its host to change the recognized Host header, or add a new Host header before original headers; 2. forward space-preceded Host headers as-is under certain conditions.
	Squid	1. for absolute-URI, rewrite to path; 2. change all recognized Host headers (except space-preceded ones) to the interpreted host, or add a new Host header after original headers; 3. forward space-preceded Host headers as-is.
Reverse Proxy	Apache	1. for absolute-URI, rewrite to path; 2. change recognized (or add a new) Host header as forwarding destination, forward as first header; 3. forward space-preceded Host headers as-is under certain conditions.
	Lighttpd	1. for absolute-URI with non-recognized schema, forward as-is; otherwise, rewrite to path;
	LiteSpeed	1. forward all recognized Host headers as-is; 2. forward space-succeeded Host headers as-is; 3. forward absolute-URI as-is.
	Squid	1. for absolute-URI, rewrite to path; 2. change all recognized Host headers (except space-preceded ones) to the interpreted host, or add a new Host header after original headers; 3. forward space-preceded Host headers as-is.
	Varnish	1. for absolute-URI with non-recognized schema, forward as-is.
CDN	Akamai	1. forward recognized space-preceded Host headers as-is; 2. forward other space-preceded Host headers as-is under certain conditions; 3. remove other recognized Host headers, add a new Host header after original headers.
	Alibaba	1. forward space-preceded Host headers and space-succeeded Host headers as-is; 2. remove all recognized Host headers, add a new Host header after original headers.
	CloudFlare	1. for absolute-URI, rewrite to path; 2. forward the first recognized Host header.
	CloudFront	1. for absolute-URI, rewrite to path; 2. remove all recognized Host headers, add a new Host header using forwarding destination before original headers; 3. forward space-preceded Host headers under certain conditions.
	Fastly	1. for absolute-URI with HTTP schema, rewrite to path; for other schemas, forward as-is.
Firewall	Bitdefender	1. for absolute-URI, forward as-is;
	ESET	1. forward the original request as-is
	Huawei	1. forward the original request as-is
	Kaspersky	1. forward the original request as-is
	OS X	1. forward the original request as-is
	PAN	1. forward the original request as-is
	Windows	1. forward the original request as-is

Table 4: Host forwarding behaviours that can potentially lead to inconsistent interpretation of host with downstream.

cern. To assess how deployed transparent caches handle requests with ambiguous `Host` headers and whether or not they make end-users vulnerable, we conducted two large-scale measurement experiments on the Internet using Flash applet. We executed our test cases by purchasing on-line Flash advertisements and obtaining a Flash hosting service on a live website, thus allowing our test Flash applet to run about one million times worldwide.

5.1 Experiments Setup

In both experiments, we set up two web servers (namely servers I and II respectively) and three domains (namely domains A, B, and C respectively). Domain A and B are hosted on server I, and domain C hosted on server II. We design 16 different test cases to study co-hosting cache poisoning and general cache poisoning, and implement all the test cases using a Flash applet.

The first 11 test cases using the three testing techniques presented in Section 3 are designed to detect transparent caches vulnerable to co-hosting cache poisoning. For each test case, we craft an ambiguous host definition using domain A and B, and send 5 requests with the ambiguous host definition to server I. We then issue one normal fetches to domain A and domain B to server I respectively. For each request, we embed a unique sequence number, which is also returned in a cache-able response by server I. If a received response has a sequence number different from the one included in a corresponding request, it indicates that the response is from a cache. The sequence number in the response also tells us which request triggers caching. We flag a vulnerable transparent cache if both two conditions hold: 1) a response to an ambiguous request is cached and the cached content is later fetched by a normal request; and 2) the forwarded request of the ambiguous request received by server I could be interpreted differently than the normal fetch that hits the cache. Figure 3 illustrates a simplified example to detect vulnerable transparent cache shown in Figure 1(d). We first send a request with ambiguous host definition and sequence number "1". Server I receives the (forwarded) request and responds with the sequence number. After sending 5 requests with the same ambiguous host, we issue a normal request with host "A" and sequence number "6". Because we see a response of sequence number "1" for the normal request, we know that a cache is present between the testing Flash and server I, and the cache identifies the ambiguous request as "A". Because the first request received by server I still has ambiguous host, we conclude that the cache is subject to co-hosting cache poisoning with

Downstream \ Upstream		Reverse Proxy							CDN								Server					
		Apache	IIS	Lighttpd	LiteSpeed	Nginx	Squid	Varnish	Akamai	Alibaba	Azure	CloudFlare	CloudFront	Fastly	Level3	Tencent	Apache	IIS	Lighttpd	LiteSpeed	Nginx	Tomcat
Transparent Cache	ATS				✓		✓		✓							✓				✓		
	Squid						✓		✓							✓						
Forward Proxy	Apache															✓						
	Squid						✓		✓							✓						
Reverse Proxy	Apache								—	—	—	—	—	—	—	—						
	Lighttpd				✓	✓			—	—	—	—	—	—	—	—				✓	✓	
	LiteSpeed	✓		✓		✓		✓	—	—	—	—	—	—	—	—	✓		✓		✓	✓
	Squid						✓		—	—	—	—	—	—	—	—						
	Varnish		✓	✓	✓	✓			—	—	—	—	—	—	—	—		✓	✓	✓	✓	✓
CDN	Akamai						✓		—													
	Alibaba				✓		✓		✓	—										✓		
	CloudFlare	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	—	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	CloudFront												—			✓						
	Fastly		✓	✓	✓	✓			✓	✓		✓	✓	—	✓			✓	✓	✓	✓	✓
Firewall	Bitdefender	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	ESET	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Huawei	✓	✓	✓	✓		✓	✓	✓		✓			✓		✓	✓	✓	✓	✓		
	Kaspersky	✓	✓	✓	✓		✓	✓	✓		✓			✓		✓	✓	✓	✓	✓		
	OS X	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓
	PAN	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓			✓	✓	✓	✓	✓	✓	✓
	Windows	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 5: "✓": upstream and downstream combinations where we can expose an inconsistent host interpretation.
"—": combinations we believe are not of practical interest.
" ": Combination is consistent in interpreting the host.

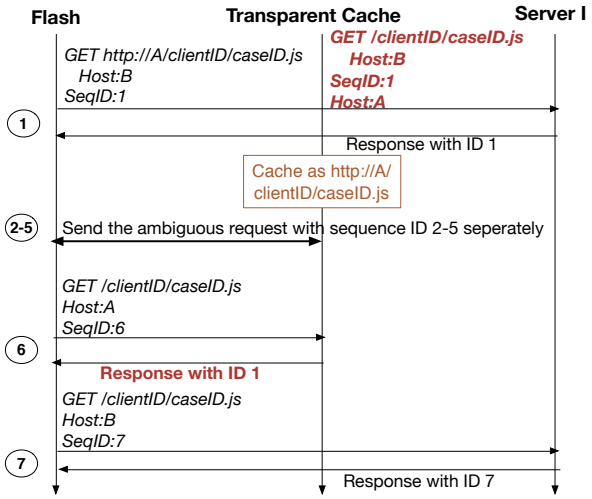


Figure 3: Illustration of detecting a transparent cache that is vulnerable to the scenario shown in Figure 1(d).

upstreams (like Akamai) accepting the first white-space preceding Host header.

We use domain A and domain C to assess if general cache poisoning is possible. Each test first issues an ambiguous request to domain A hosted on server I for 5 times, followed by two normal requests to domain C hosted on server II. Sequence numbers in requests and responses are also used to detect caching behaviour. If a normal request is responded with a cached content corresponding to a previous ambiguous request, we know that a cache is present, which caches a response from server I as a resource of domain C. Therefore we conclude that the cache is vulnerable to general cache poisoning. We design another 5 test cases to uncover such transparent caches.

The testing starts when a client browser or other runtime loads the Flash applet. We took great care in constructing our Flash applet to ensure there are no side effects beyond caching our own elements. Although our requests are non-compliant, they should not trigger any memory error or other conditions. And we do not attempt to perform any other activity beyond simply checking whether we can ambiguously cache data involving our own domains. For privacy, we collected only properties typically disclosed by browsers when viewing web pages (e.g. request headers, and external IP addresses).

5.2 Result Analysis

We conducted two experiments using the same Flash applet. The first experiment was from December 11 2015 to December 31 2015. We brought 1.5 million advertising impressions with about $110 on the Bit-torrent PC client uTorrent, which distributes Flash advertisements as part of the revenue model. Due to a server configuration change, we discarded approximately 100K impressions. For the other 1.4 million impressions, we received testing results from 971,343 unique IP addresses, covering 228 countries and 12,631 different ASes. To increase the coverage of measurement in China, we also hosted the testing Flash on a Chinese website from March 11 2016 to March 31 2016. In the second experiment, we received testing results from 175,375 unique IP addresses, mostly in China. Figure 4 shows the geographical distribution of involved clients in two experiments.

In the first experiment, we identified transparent caches from testing sessions of 16,168 IP addresses. Among them, 15,677 (96.9% of transparent caches) different IPs are vulnerable to at least one form of our cache poisoning attacks. 13,184 IP addresses are vulnerable to co-hosting cache poisoning, 4,259 IPs are vulnerable to general cache poisoning, and some of them are vulnerable to both.

The second experiment detected that 1,331 (96.7%) out of 1,376 IP addresses behind transparent caches can be affected by co-hosting cache poisoning. 6 are vulnerable to both co-hosting cache poisoning and general cache poisoning.

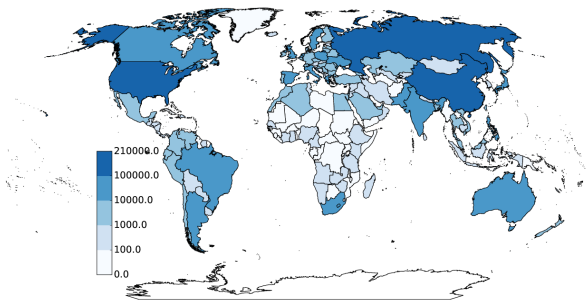


Figure 4: The geographical distribution of client IP addresses involved in two experiments.

Server	Vuln IP#	Reverse Proxy	Vuln IP#	CDN	Vuln IP#
Apache	9075/201	Apache	9075/201	Akamai	12337/416
IIS	9075/200	IIS	9075/200	Alibaba	9749/202
Lighttpd	9075/199	Lighttpd	9075/199	Azure	9075/199
LiteSpeed	10319/199	LiteSpeed	10319/199	CloudFlare	9749/202
Nginx	9749/202	Nginx	9749/202	CloudFront	9749/202
Tomcat	9748/202	Squid	11378/415	Fastly	9091/201
		Varnish	9068/199	Level3	9711/200
				Tencent	9843/211

Table 6: The amount of IP addresses vulnerable to co-hosting cache poisoning involving different upstreams in the first and second experiment.

A transparent cache vulnerable to co-hosting cache poisoning may be exploited when connecting to one or more specific co-hosting upstream servers. We looked into the parsing, interpreting, and forwarding behaviours of the vulnerable transparent caches to find their potential "cooperating" upstreams in the 21 implementations presented in Table 5. Table 6 shows the number of vulnerable IP addresses for particular upstream configurations. From the table, we can see that the number of potentially vulnerable IP addresses is largest when the upstream is Akamai (CDN), Squid (Reverser Proxy), LightSpeed (Reverse Proxy and Server) and Tencent (CDN). The general reason is that they are more liberal with requests with malformed hosts. They do not reject multiple Host headers. They also accept space before or after Host header, which is often transparently ignored by an in-path proxy. Thus, they are more likely to be inconsistent with others and be attacked.

To look deeper at the number of vulnerable IP addresses across different countries, we listed the top 10 countries in which vulnerable IP addresses in two experiments are distributed, as shown in Figure 5. In this process, we can see that India has the largest number of vulnerable IP addresses, closely followed by the Philippines and Brazil. Apart from that, the amount of IP addresses vulnerable to co-hosting cache poisoning is larger than that of general cache poisoning in most countries, except Philippines. Combined with Table 7, we observed that most vulnerable IP addresses in some countries (such as India, Philippines, China and New Zealand) are concentrated in several ASes.

One limitation with our testing is that our Flash applet is primarily run in a Windows BitTorrent client that is used by the advertising service we purchased. Since most users likely do not run BitTorrent clients over usage-billed and bandwidth limited cellular networks, our tests primarily cover transparent caches on the fixed Internet, with very limited

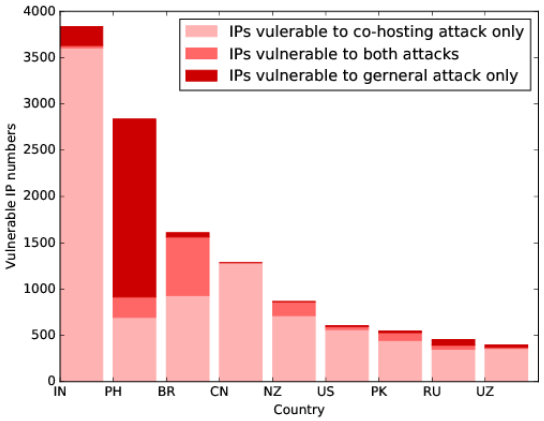


Figure 5: Top 10 vulnerable IPs sorted by Country

Country	ASN	Organization	#
PH	9299	Philippine Long Distance Telephone	2396
IN	23860	Alliance Broadband Service	1234
IN	24309	Atria Convergence Technologies	1013
CN	56046	China Mobile	692
CN	9808	China Mobile	476
PH	132199	Globe Telecom	429
NZ	9790	CallPlus Services Limited	410
NZ	7657	Vodafone NZ Ltd.	377
US	3651	Sprint	317
SA	35819	Etihad Etisalat Company (Mobily)	302

Table 7: Top 10 vulnerable IPs sorted by ASN

coverage of cellular network from the mobile users of our Flashing hosting website.

Regardless of visibility concerns, this survey does confirm an unfortunate fact: almost all caches we measured were vulnerable to at least one cache poisoning scenario.

5.3 Case Study

In the process of analysing the measurement results, we found several vulnerable IP addresses located in National University of Singapore (NUS). These IP addresses are vulnerable to both co-hosting cache poisoning and general cache poisoning. To validate our results, we performed our test cases from a Planetlab node in NUS campus network manually, and verified with browser to confirm that NUS campus network deployed commercial transparent caches and was indeed vulnerable to the two cache poisoning attacks. We reported these vulnerabilities to Computer Center of NUS, and got confirmation from them.

6. NOTIFICATION AND RESPONSE

We made attempt to contact both CERT/CC and individual vendors. CERT/CC has acknowledged our report and assigned a VU number (#916855) to track this problem. Currently we have successfully contacted 13 individual vendors, and their responses are summarized in below.

6.1 Cache Poisoning Attacks

Squid: Our report to the Squid team resulted in two public security update advisories (CVE-2016-4553 [24] and CVE-2016-4554 [25]). For the general cache poisoning attack affecting both Squid3 and Squid4, the Squid team evaluated it as the highest level (Blocker) of security vulnerability and fixed it in version 3.5.18 and 4.0.10. For the co-

hosting attack, they said the vulnerability was introduced into Squid 1.0 in 1996 and modified Squid3 to not accept space-preceding Host headers (versus concatenating it with the preceding header). However, they are not considering fixing the problem in Squid4, since Squid4 does not accept but simply ignore space-preceding Host headers. We pointed out that Squid4 still forward space-preceding Host headers, which may be accepted by an upstream. They suggest that it is up to an upstream service provider (such as Akamai) to make their own implementation compliant with RFC 7230. They also suggest that our exploitation methods could also be applied to some other headers (such as Content-Length) to re-enable related attacks such as HTTP request smuggling attacks.

Akamai: We reported this problem to Akamai, which has confirmed that our exploitation methods are effective in cache poisoning. They mentioned that our report sparked considerable internal discussion and debate. They have deployed a solution to defend against this problem.

Alibaba: Alibaba confirmed the attacks in our report and have modified their servers to mitigate these attacks immediately after our report.

Tencent: Tencent confirmed that the attacks in our report were valid and have fixed them at this time.

Apache Traffic Server: Apache Traffic Server acknowledged and confirmed the attack in our report. But they did not tell us whether they will fix it.

6.2 Filtering Bypass

Palo Alto Networks: Palo Alto Networks took our report seriously, and invited us to have a face-to-face discussion. They said the diversity behaviours of different web servers were out of their expectation. They expressed concerns of false positives for enforcing a strict HTTP compliance, because of the diversity of real world traffic. They are willing to add extra options in their future release for customers to determine whether or not to block the ambiguous requests we reported.

Huawei: Huawei immediately formed a team to work on this issue and confirmed the problem. They also invited us to a face-to-face meeting to discuss it further. They would provide options for their customers to enforce strict RFC 7230 compliance.

ESET: ESET confirmed the attacks and were fixing it. They offered several T-shirts and a hard copy of the acknowledgement as a token of gratitude.

CloudFlare: CloudFlare acknowledged our report, and had a detailed discussion with us about its implications. They are working a fix at this time.

Fastly: Fastly discussed with us, and acknowledged that the problem could be an issue under certain conditions.

Kaspersky: Kaspersky confirmed the exploits to bypass their parental control feature. But they think it is not critical because it requires specific software installation which is not available for a child at properly configured OS.

Microsoft: Microsoft thinks it is a product-related bug rather than a security vulnerability.

7. DISCUSSION

7.1 Mitigation

Strictly speaking, this is an implementation problem rather than a specification problem. While RFC 2616 has some ambiguities in the handling of Host, RFC 7230 is generally strict and clear. Therefore, we recommend that vendors including both downstream and upstream, fully comply with RFC 7230 to avoid problems arising due to inconsistent Host interpretations. Per RFC 7230, the correct approach is to treat multiple Host headers and whitespace around field names as errors.

We have seen false positive concerns from some firewall vendors. We suggest that firewall vendors with such concerns could provide options for their customers to enforce strict RFC 7230 compliance, rejecting or alerting any invalid requests. As we believe that multiple host ambiguities should not be present in any benign request, we encourage vendors with false positive concerns to collect real-world statistics. If the real-world data support our hypothesis, vendors should enable full compliance as default.

Apart from the HTTP implementations studied in this paper, these problems related to `Host` headers could also affect other systems and/or manifest in other forms. We recommend that developers of any deployed system that processes HTTP requests with a notion of an associated host should review their implementations with this threat in mind.

We anticipate a long period for the deployed devices to get patched, because of the prevalence of affected systems. Websites can mitigate the effects of vulnerable transparent caches by deploying HTTPS with HTTP Strict Transport Security (HSTS) [9], preferably with preloading. HTTPS with HSTS prevents clients from issuing plaintext HTTP requests, therefore avoids the clients being attacked by poisoned transparent caches because the caches are usually not capable to intercept encrypted traffic.

To aid in identifying Host of Troubles issues, we have consolidated the attack techniques into an online checking tool, [3] which helps client users and ISP operators to automatically evaluate whether their network are vulnerable to the cache poisoning attacks we found.

7.2 Protocol Design and Implementation

Our study underscores an unfortunate fact: most HTTP implementations lack full compliance with RFC 7230. While some factors, such as backward compatibility, may contribute to this fact, our experience suggests that the presentation of RFC 7230 regarding how to treat Host headers could be improved. In particular, we argue for the benefit of providing a thoroughly reviewed reference implementation, for two reasons. First, currently Host-related rules appear in multiple places; a reference implementation would help to aggregate them together for consideration in a single place. Second, specifications written in natural language inevitably introduce ambiguities, due to either the wording itself, or from the incomplete understanding of implementers. A reference implementation would help address both considerations.

Some implementation ambiguities we examined relate to the protocol design of HTTP. In particular, the redundant semantics of Host and the host component in URLs introduces the possibility of ambiguities. While a strict specification can clarify and regulate protocol fields with redundant semantics, problems often arise when implementations do not fully comply with the specification, as demonstrated in this study. In general, when designing protocols we should be careful to avoid introducing opportunities for overlapping and potentially conflicting semantics in protocol fields,

[3] https://hostoftroubles.com/online-checker.html

rather than attempting to resolve such issues by specification rules.

Another protocol design perspective highlighted by this problem is that the correct origin and context association of HTTP messages depends on consistent states between multiple parties, and does so without incorporating additional error-detection/recovery mechanisms. Such design can prove fragile in the face of attacks that exploit ambiguities caused by implementation imprecision. Some protocol enhancements could make HTTP more resilient to origin confusion attacks. For example, adding a cryptographically verifiable origin to HTTP responses would help to detect potential origin confusions. However, the effectiveness of such enhancements would still rely on correct implementation.

Our study highlights the gap between protocol specification and implementation, especially when a protocol keeps evolving. Community efforts beyond IETF working groups focused on standardization would help to reach more implementers and to increase the awareness of important protocol changes. For example, for a number of years the IETF had a working group (TCP-IMPL) chartered specifically to discuss TCP implementation issues, rather than to standardize aspects of TCP.

Finally, the Host of Troubles vulnerabilities highlight a fundamental tension underlying Postel's robustness principle. Protocol implementations being liberal in what they accept has great utility in facilitating unfettered connectivity between trusted parties; but in adversarial situations, it opens the floodgates to myriad potentially exploitable ambiguities. While protocol designers may be aware of these limitations of the robustness principle [26], our study shows that implementers still largely overlook its hazardous implications.

8. RELATED WORK

Some have developed abusive uses of untrustworthy Host header to exploit insufficient input validation in web applications [1, 12], The consequence can be phishing, cross-site scripting, etc. The cause of these attacks is that web applications misuse host-related variables passed by their frontend HTTP implementations that parse and interpret raw HTTP requests. Broadly speaking, these attacks are also exploitations of semantic inconsistency of the HTTP host. The difference between our work and these attacks is that in our work the semantic inconsistency is caused by discrepancies in parsing and interpreting of raw HTTP request, while in those attacks, the semantic inconsistency is caused by different assumptions of host-related variables between a caller and a callee. Kettle also briefly sketched different handling of multiple Host headers in different implementations [12]. Our work fleshes out his sketch with a variety of multiple host ambiguities and an in-depth empirical study.

Inconsistent host interpretations between different parties can have disastrous consequences, because hosts provide the basis in HTTP environments for isolating different security domains. Delignat-Lavaud and Bhargavan showed that host confusion and consequent isolation violations can also occur due to operation and configuration defects, especially in environments involving HTTPS [3].

The general cache poisoning attack we found has roots in a particular implementation problem faced by transparent caches. Upon receiving a request, a transparent cache needs to decide whether to directly forward the request to the destination IP address of its underlying TCP connection, or to initiate a new connection to the host in the request. If the transparent cache chooses the former, and caches the response without further checks, it becomes subject to general cache poisoning by requests that specify arbitrary hosts. The latter choice, unless coupled with further protection, can result in the abuse of the transparent cache's IP address to probe internal websites that can only be reached through the transparent cache. The latter problem has been reported as CERT VU#435052 [7]. Huang et al [10] used Adobe Flash and a Java applet to measure the prevalence of both vulnerabilities in the real world. Squid chose the former implementation approach, with an additional consistency check comparing the destination IP address with the claimed host to avoid cache poisoning. But the inconsistent notion of host within its internal modules allows us to bypass this check using requests with multiple hosts.

Our Host of Troubles attacks belongs to a family of "semantic gap" attacks [11] that exploit the difference in interpreting an object by two or more parties. Some other semantic gap attacks have been identified in HTTP implementations, such as HTTP "request smuggling" attacks [15]. The attacks we found differ from request smuggling in that our attacks exploit discrepancies in the host definition of one request to create host confusion, while request smuggling takes advantage of implementation differences in Content-Length to induce inconsistencies in request-response association. In general, semantic gap attacks are difficult to enumerate, and identifying one vector does not necessarily shed light on other potential vectors. Our study shows that the defenses against request smuggling attacks do not help prevent Host of Troubles attacks, despite their conceptual similarity. In fact, some vendors expressed concerns that the use of whitespace in Host of Troubles attacks may also apply to Content-Length manipulation to re-enable request smuggling attacks. Another form of semantic gap attack, HTTP Evader [27], exploits ambiguities in parsing responses to evade anti-virus firewalls, and Ristic presented a number of techniques to bypass web application firewall (WAF) rules [22]. Other examples include manipulations of IP packets [20, 21, 8, 29, 13, 16], files [11, 17, 18], and other operating system resources [23].

Vulnerable proxies such as transparent caches are the most significant threat exposed in this work. Weaver et al [30] used Netalyzr [14] to discover the presence of proxies on the Internet. Their results show that a significant fraction of end user HTTP traffic goes through proxies. Xu et al [31] studied a number of behaviors of web proxies in cellular networks, including caching, content rewriting, and redirection, among others. Their results indicate that all four US carriers they tested deploy web proxies, albeit with different behaviors. Both studies could serve evidence that the real world impact by Host of Troubles could be significant due to the prevalence of proxy deployment.

9. CONCLUSION

While Postel's robustness principle can greatly facilitate unfettered connectivity between trusted parties, the ambiguities it tends to introduce in Internet implementations can prove detrimental to security in adversarial environments. We present a class of attacks, "Host of Troubles", that leverage ambiguous interpretations of HTTP's Host header to enable cache poisoning attacks and security policy bypasses.

The root cause lies in implementations that, contrary to RFC 7230, inconsistently parse and interpret the Host header and related information in request-URIs.

Attackers can exploit this problem by carefully crafting HTTP requests with ambiguous host information, inducing inconsistent interpretations between two parties, with varying consequences depending on the particular scenario. We examined 33 popular HTTP implementations and found a number of interpretation inconsistencies that attackers can exploit, generally by chaining together incompatible interpretations. By conducting two large-scale measurements, we show that around 97% of users served by transparent caches are affected by the cache poisoning attacks we found.

Our work underscores the importance of standard compliance. It also shows the consequence of implementations guided by the robustness principle without also incorporating thorough security considerations.

10. ACKNOWLEDGMENTS

We especially thank Ouyang Xin, Wei Xu, Zhi Xu, Jiangxia Liu from Palo Alto Networks, Shiguang Li from Huawei for valueable discussion. We also thank Amos Jeffries from Squid, Nick Sullivan and Evan Johnson from CloudFlare, Daniel McCarney and Jonathan Foote from Fastly and Mike Kun from Akamai for their helpful comments and feedback. We are grateful to the anonymous reviewers, and Jinjin Liang, Xiaofeng Zheng, Baojun Liu, Kun Du, and Kai Zhang for suggestions and feedback. This work was funded by Tsinghua National Laboratory for Information Science and Technology (TNList) Academic Exchange Foundation, Natural Science Foundation of China (grant #61472215) and was also partially supported by the US National Science Foundation under grant CNS-1237265, and by generous support from Google and IBM. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of their employers or the funding agencies.

11. REFERENCES

[1] Bueno, C. HTTP Cache Poisoning via Host Header Injection. http://carlos.bueno.org/2008/06/host-header-injection.html, June 2008.

[2] Chen, J., Jiang, J., Zheng, X., Duan, H., Liang, J., Li, K., Wan, T., and Paxson, V. Forwarding-Loop Attacks in Content Delivery Networks. In *Proceedings of the 23st Annual Network and Distributed System Security Symposium (NDSS'16)* (2016).

[3] Delignat-Lavaud, A., and Bhargavan, K. Network-based origin confusion attacks against https virtual hosting. In *Proceedings of the 24th International Conference on World Wide Web* (New York, NY, USA, 2015), WWW '15, ACM, pp. 227–237.

[4] Durumeric, Z., Wustrow, E., and Halderman, J. A. ZMap: Fast Internet-wide Scanning and Its Security Applications. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)* (Washington, D.C., 2013), USENIX, pp. 605–620.

[5] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585.

[6] Fielding, R., and Reschke, J. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230 (Proposed Standard), June 2014.

[7] Giobbi, R. Vulnerability Note VU#435052: Intercepting Proxy Servers may Incorrectly Rely on HTTP Headers to Make Connections. http://www.kb.cert.org/vuls/id/435052, February 2009.

[8] Handley, M., Paxson, V., and Kreibich, C. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *USENIX Security* (2001).

[9] Hodges, J., Jackson, C., and Barth, A. HTTP Strict Transport Security (HSTS). RFC 6797 (Proposed Standard), Nov. 2012.

[10] Huang, L.-S., Chen, E. Y., Barth, A., Rescorla, E., and Jackson, C. Talking to Yourself for Fun and Profit. *Proceedings of W2SP* (2011), 1–11.

[11] Jana, S., and Shmatikov, V. Abusing File Processing in Malware Detectors for Fun and Profit. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2012), SP '12, IEEE Computer Society, pp. 80–94.

[12] Kettle, J. Practical HTTP Host Header Attacks. http://www.skeletonscribe.net/2013/05/practical-http-host-header-attacks.html, May 2013.

[13] Korhonen, E. Advanced Evasion Techniques - Measuring the Threat Detection Capabilities of Up-to-Date Network Security Devices. *Master's Thesis* (08 2012).

[14] Kreibich, C., Weaver, N., Nechaev, B., and Paxson, V. Netalyzr: Illuminating The Edge Network. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement* (2010), ACM, pp. 246–259.

[15] Linhart, C., Klein, A., Heled, R., and Orrin, S. HTTP Request Smuggling. *Computer Security Journal 22*, 1 (2006), 13.

[16] Niemi, O.-P., and Levomäki, A. Evading Deep Inspection for Fun and Shell. *Black Hat USA* (2013).

[17] Oberheide, J., Bailey, M., and Jahanian, F. PolyPack: an Automated Online Packing Service for Optimal Antivirus Evasion. In *Proceedings of the 3rd USENIX conference on Offensive technologies* (2009), USENIX Association, pp. 9–9.

[18] Porst, S. How to Really Obfuscate your PDF Malware. *RECON, July* (2010).

[19] Postel, J. Transmission Control Protocol. RFC 793 (INTERNET STANDARD), Sept. 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[20] Ptacek, T. H., and Newsham, T. N. Insertion, Evasion, and Denial of service: Eluding Network Intrusion Detection. Tech. rep., DTIC Document, 1998.

[21] Puppy, R. F. A Look at Whisker's Anti-IDS Tactics. *Online* (12 1999).

[22] Ristic, I. Protocol-level evasion of web application firewalls. *Black Hat USA* (2012).

[23] Su, Z., and Wassermann, G. The Essence of Command Injection Attacks in Web Applications. In *ACM SIGPLAN Notices* (2006), vol. 41, ACM, pp. 372–382.

[24] Team, S. Squid Proxy Cache Security Update Advisory SQUID-2016:7. http://www.squid-cache.org/Advisories/SQUID-2016_7.txt, May 2016.

[25] Team, S. Squid Proxy Cache Security Update Advisory SQUID-2016:8. http://www.squid-cache.org/Advisories/SQUID-2016_8.txt, May 2016.

[26] Thomson, M. The Harmful Consequences of Postel's Maxim. https://tools.ietf.org/html/draft-thomson-postel-was-wrong-00, March 2015.

[27] Ullrich, S. HTTP Evader - Automate Firewall Evasion Tests. http://noxxi.de/research/http-evader.html.

[28] Vissers, T., Van Goethem, T., Joosen, W., and Nikiforakis, N. Maneuvering Around Clouds: Bypassing Cloud-based Security Providers. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 1530–1541.

[29] Vutukuru, M., Balakrishnan, H., and Paxson, V. Efficient and Robust TCP Stream Normalization. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on* (2008), IEEE, pp. 96–110.

[30] Weaver, N., Kreibich, C., Dam, M., and Paxson, V. Here Be Web Proxies. In *Proceedings of the 15th International Conference on Passive and Active Measurement* (New York, NY, USA, 2014).

[31] Xu, X., Jiang, Y., Flach, T., Katz-Bassett, E., Choffnes, D., and Govindan, R. Investigating Transparent Web Proxies in Cellular Networks. In *Passive and Active Measurement* (2015), Springer, pp. 262–276.