Who's Breaking the Rules? Studying Conformance to the HTTP Specifications and its Security Impact

Jannis Rautenstrauch
CISPA Helmholtz Center for Information Security
jannis.rautenstrauch@cispa.de

Ben Stock
CISPA Helmholtz Center for Information Security
stock@cispa.de

ABSTRACT

HTTP is everywhere, and a consistent interpretation of the protocol's specification is essential for interoperability and security. In 2022, after more than 30 years of evolution, the core HTTP specifications became an Internet Standard. However, apart from anecdotal evidence showing that HTTP installations violate parts of the specifications, no insights on the state of conformance of deployed HTTP systems exist. To close this knowledge gap, we systematically analyze the conformance landscape of HTTP systems with a focus on the potential security impact of rule violations.

We extracted 106 falsifiable rules from HTTP specifications and created an HTTP conformance test suite. With our test suite, we tested nine popular web servers and 9,990 live web hosts. Our results show that the risk for security issues is high as most HTTP systems break at least one rule, and more than half of all rules were broken at least once. Based on our findings, we propose improvements, such as more conformance testing and less reliance on the robustness principle and instead explicitly defining error behavior.

CCS CONCEPTS

• **Security and privacy → Web protocol security**.

KEYWORDS

HTTP, Measurement, Specifications, Rule Violations

ACM Reference Format:
Jannis Rautenstrauch and Ben Stock. 2024. Who's Breaking the Rules? Studying Conformance to the HTTP Specifications and its Security Impact. In *Proceedings of the 19th ACM ASIA Conference on Computer and Communications Security (ACM ASIACCS 2024), July 1–5, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/TBA

1 INTRODUCTION

The Web and its core communication protocol Hypertext Transfer Protocol (HTTP) are everywhere: Phones, Computers, Cars, Watches. HTTP is notorious for being complicated, having evolved over many years and versions. The latest specifications are distributed over dozens of documents with hundreds of features and thousands of lines. Additionally, modern HTTP communication involves many participants, such as browsers, crawlers, proxies, load balancers, and origin servers. They all need to follow the same specifications and agree on the same interpretation. Otherwise, interoperability problems, performance issues, semantic gap security threats, or inconsistent protection against attacks can occur.

Evidence from browsers shows that different implementations often diverge in their interpretation and implementation of specifications [40, 49, 15]. However, apart from anecdotal evidence and analysis of singular features, e.g., caching [51], no common knowledge of web servers' conformity to specifications exists. The main reason for this lack of knowledge is that no commonly agreed-on test suite for HTTP conformance exists. Browsers have created the Web Platform Tests [84], but nothing similar for HTTP exists. REDbot [56], an HTTP linter, does not suffice because it refers to the previous RFC generation of HTTP, misses many features such as HTTP/2 support (nowadays used by up to 66% of websites [63]), and has no security focus.

To close this research gap, we investigate the following four research questions: (1) *What is the state of HTTP conformance in the wild?*; (2) *Does HTTP conformance vary between popular and less popular websites?*; (3) *Which consequences does non-conformity to the HTTP specifications have?*; and (4) *How could HTTP specification conformance be improved?*

To aid in that, we created an HTTP/1 and HTTP/2 security-focussed conformance test suite by systematically analyzing HTTP specification-related documents, extracting falsifiable rules for requirements and recommendations, and implementing test cases for each rule. We then extensively analyzed HTTP rule violations on nine local servers and 10,000 websites. The results highlight that HTTP conformance is far from perfect, as over half of our rules are violated at least once, and most websites violate at least one rule, while the local servers violate between six and ten rules.

This new test suite, which we release with the paper [60], allows us to make the following contributions:

- We create a methodology to test for HTTP conformance of responses and open-source a large set of conformance tests for HTTP rules (Section 3).
- We perform the first web-scale evaluation of HTTP specification conformance for both local servers and 10,000 general websites (Section 4).
- We discuss potential negative security implications of violations from our measurement results (Section 5).
- We identify causes of lacking conformance and provide a set of *recommendations* to have less divergence between specifications and implementations (Section 6).

ACM ASIACCS 2024, July 1–5, 2024, Singapore, Singapore

ACM ISBN TBA
https://doi.org/TBA

2 BACKGROUND

HTTP is the fundamental application protocol of the web. From its initial purposes as a simple document-sharing system in 1989 [10], the protocol developed over many iterations to a massive complexity, allowing native-like applications in the browser. In this section, we explain the history of HTTP, the request-response pattern of HTTP, how different participants should conform to the specifications, and which (security) issues can arise from non-conformance.

2.1 History of HTTP

The original version of HTTP, envisioned in 1989 and implemented in 1991, only had minimal functionality. This version was never formally specified but is nowadays known as HTTP/0.9 [9]. In the following years, many extensions were implemented and tested to allow for more complex applications, e.g., requiring authentication. These efforts resulted in an after-the-fact informational specification of HTTP/1.0 in 1996 [54]. To unify server behavior and fix many of the discovered issues, the first HTTP/1.1 specification was published only one year later as a proposed standard [26]. Two years later, the HTTP/1.1 specifications were updated and upgraded to a draft standard [55].

After being stable for a long time, the HTTP/1.1 specifications were updated in 2014 [30] to clarify ambiguities that came up and deprecate certain features such as *header folding over multiple lines* that turned out to be dangerous. As HTTP always was a plaintext protocol, several performance issues existed. So, an updated version using a binary protocol was introduced in 2015. The new version is called HTTP/2 and uses almost the same semantics but a different transmission over the wire [8]. In 2022, the HTTP specifications finally reached the status of an Internet Standard, the highest specification grade. In the update, the HTTP semantics [27] were separated from the underlying transmission over the wire protocol, such as the updated HTTP/1.1 [29], HTTP/2 [72] and newly introduced HTTP/3 [11] protocols.

However, the upgrade to an official Internet Standard was neither widely published nor recognized. Many HTTP proxies, servers, and clients still refer to RFC 7230 or even RFC 2616. For example, the codebase of Apache [31] returns four matches for RFC 9110 but ten matches for RFC 7230 and even 66 for RFC 2616. Similarly, the codebase of node.js [32] matches RFC 9110 two times, whereas RFC 7230 matches eleven times and RFC 2616 matches eight times. Popularly used third-party documentation has the same issues. The *Evolution of HTTP* [18] article of MDN still refers to RFC 7230 even though it was updated in 2023.

In addition to the core HTTP specifications, many additional features, such as specific headers or request methods, were defined in other RFCs and documents, such as in the Fetch [80] or HTML living standard [81]. All in all, HTTP became a complex topic. New developers wanting to create a conformant HTTP processor would need to read hundreds of pages distributed over dozens of documents and often incorrectly start with RFC 2616 or third-party resources based on RFC 2616 instead of RFC 9110.

2.2 Request and Response Pattern

Regardless of the version of HTTP, its encryption status, and its underlying transport protocol, HTTP always uses a request/response pattern. For brevity and understanding, we describe an HTTP/1.1 exchange; other versions use a roughly equivalent form.

```
GET /test/ HTTP/1.1
Host: conformance.spec
Accept-Language: en
```

(a) HTTP/1.1 request

```
HTTP/1.1 200 OK
Date: Wed, 17 August 2023 14:28:02 GMT
Server: RuleBreaker
Content-Length: 29769
Content-Type: text/html

(29,769 bytes of the requested web page)
```

(b) HTTP/1.1 Response

Figure 1: HTTP request and response

Figure 1 shows an example request/response pair. A client sends a request, consisting of the request line specifying the method *GET*, the path */test/*, and the HTTP version *HTTP/1.1*, to a server. Subsequently, the client can add request headers; the Host header *must* be present according to RFC 9110 [28], while all other request headers are optional. The headers are then terminated by an empty line, followed by an optional message body (omitted from the example).

A server parses received requests, processes them, and then returns an HTTP response to the client. A response first consists of the status line, containing the HTTP version *HTTP/1.1*, the status code *200*, and optionally of a reason phrase *OK*. In line with request headers, the server can also specify an arbitrary number of response headers, some of which are mandatory. Which headers are required depends on the returned status code, the capabilities of the server, and the request method and headers. Finally, the response can carry a body (here 29,769 bytes), which again is separated from the headers by a blank line.

2.3 HTTP Participants and Conformance

Modern HTTP communication does not only consist of a client performing a request and a server generating a response as we have described in Figure 1. Instead, most requests to websites pass through an HTTP processing chain containing a plethora of intermediaries such as proxies or caches. These intermediaries can alter requests and responses or decide not to pass them on and generate their own response, e.g., return a cached element or an error.

Most HTTP-related specifications follow RFC 2119 [12] to specify requirements and recommendations. Furthermore, the core HTTP specification states that "An implementation is considered conformant if it complies with all of the requirements associated with the roles it partakes in HTTP." [28]. Many rules apply to all participants. For example, the *Allow* header is mandatory if the status code is *405*, and generated headers must fulfill their syntactic requirements as specified by their ABNF grammars. However, some rules only affect certain participants or depend on their protocol role. One example is that while generating headers, senders must strictly adhere to a header's ABNF; however, receivers should apply laxer parsing rules to allow for common errors.

2.4 (Security)-Issues of Nonconformity

Not following specifications to the letter, i.e., breaking HTTP rules, can have negative consequences. In general, these consequences highly depend on the exact rule violated and also how the rule is violated. In the following, we survey known (security) issues related to incorrect HTTP messages.

First, non-conformance could mean that certain (mandatory) features are not supported. The missing features could lead to issues for end-users when accessing the server in a browser. Also, it could break functionality or confuse other tools, such as web crawlers or automated testing tools.

Second, violations of many rules can negatively impact performance on the web. For example, the *Date* header is required if a sender has a clock; without that header, caching is impacted. Also, if persistent connections do not work, connections to a site will be slow.

Even worse is that differing interpretations of the HTTP specification and failing to parse requests and responses strictly can lead to so-called semantic gap attacks [14, 65], such as Host of Troubles [17], HTTP Request Smuggling [42, 41], Cache-Poisoned Denial-of-Service [52], or Web Cache Deception [50]. The starting point of such attacks is often that at least one participant breaks an HTTP rule motivating a general study on HTTP specification conformance.

Lastly, if HTTP participants regularly receive invalid requests and responses, they often start writing workarounds to be able to handle them. The specifications explicitly allow such workarounds: "A recipient MAY employ such workarounds while remaining conformant to this protocol if the workarounds are limited to the implementations at fault. For example, servers often scan portions of the User-Agent field value, and user agents often scan the Server field value, to adjust their own behavior with respect to known bugs or poorly chosen defaults." [28] However, such workarounds can lead to even more issues. First, more workarounds emerge over time, leading to a code complexity explosion. Also, as updates to the violating software are rare, the workarounds must stay forever and might even be persisted in the specifications. Even if the software gets updated, the workaround might stay in place, leading to compatibility issues with the new version. For example, many websites did not serve security headers such as HSTS or CSP to legacy browsers based on the user-agent header. As Firefox on iOS until version 96.0 used a different versioning scheme than their desktop counterpart, users using Firefox on iOS were incorrectly recognized as legacy clients and received less secure responses [62]. Due to the same issue, browsers often implement error tolerance where the browsers are trying to accept way more inputs than specified by a grammar or trying to repair invalid values automatically [38]. As a result, various issues such as mutationXSS [39], browser fingerprinting [2], and inconsistent security header behavior between browsers [15] arise.

3 METHODOLOGY

This work studies the landscape of HTTP conformance of deployed HTTP systems. It is not possible to show that an implementation is conformant, but it is possible to show that an implementation is non-conformant by demonstrating that it breaks individual rules of HTTP; an implementation that breaks such rules can be classified as non-conformant. To create such rules, we performed a systematic analysis of *the* HTTP specification and extracted falsifiable properties. In the following, we explain which documents we considered, how we extracted falsifiable rules, and how we tested whether real HTTP systems follow these rules.

3.1 HTTP Specification(s) and Rules

There is no single comprehensive HTTP specification document. Instead, HTTP-related specifications are distributed over many documents issued by several organizations, such as the IETF, W3C, or WHATWG. The core of HTTP is defined by the recently updated RFCs that describe the semantics of HTTP (9110), caching in HTTP (9111), and different HTTP versions (9112-9114). These RFCs build on earlier versions that are now obsolete. In addition, other documents specify many additional features related to HTTP.

Table 1 lists all specification documents we considered along the number of extracted rules. The table also shows the number of broken rules per document presented in Section 4. We consider the core HTTP spec documents RFC 9110-9111, HTTP/1.1 (RFC 9112), and HTTP/2 (RFC 9113). In addition, we consider two RFCs defining important HTTP features that precede the current generation of HTTP specifications but were not obsoleted by them: Cookies (RFC 6265/State Management) and Patch Method (RFC 5789). Further, we include RFC 6797 and five other documents specifying (browser) security features activated by HTTP headers.

3.1.1 Rule Criteria. After having identified the relevant documents, we extract falsifiable rules from them. Here, we leverage the MUST and SHOULD language specified by RFC 2119 [12]. Rules that would be classified Optional (MAY in RFC) are not considered, as the specification does not pose any binding guidelines for implementation, and implementations can differ widely for good reasons.

While rules for all participants of the HTTP protocol, such as user agents (i.e., clients), origin servers (i.e., websites), and intermediaries (e.g., caches or load balancers) exist, we focus on rules that are valid on all responses regardless of which participant produced the response. This selection criterion enables us to test arbitrary websites as we do not need to know which participant of a complex HTTP processing chain generated or modified the final response. We do not test HTTP clients or take rules applying to requests into account as we want to study the landscape of deployed HTTP systems. We leave the testing of HTTP clients open for future work.

Many of the rules for HTTP participants are not falsifiable in a black-box model. For example, to falsify the following rule "A recipient MUST ignore the If-Modified-Since header field if the resource does not have a modification date available", we would need to know whether the resource has a modification date and whether the header was ignored, neither of which is realistic from external observation. Thus, we focused on rules that are clearly falsifiable by only analyzing responses received after sending requests to servers.

3.1.2 Rule Extraction. Based on the above criteria, we extract rules for HTTP participants that can be falsified by analyzing responses after sending probe requests to servers, regardless of whether additional intermediaries are involved.

Name	Title	Status	Organization	Date	#Rules	#Broken Rules
RFC 5789	PATCH Method for HTTP	Proposed Standard	IETF	March 2010	2	2
RFC 6265	HTTP State Management Mechanism	Proposed Standard	IETF	April 2011	4	4
RFC 6797	HTTP Strict Transport Security (HSTS)	Proposed Standard	IETF	November 2012	5	4
RFC 9110	HTTP Semantics	Internet Standard	IETF	June 2022	55	38
RFC 9111	HTTP Caching	Internet Standard	IETF	June 2022	7	4
RFC 9112	HTTP/1.1	Internet Standard	IETF	June 2022	10	6
RFC 9113	HTTP/2	Proposed Standard	IETF	June 2022	5	1
CSP	Content Security Policy Level 3	Working Draft	W3C	February 2023	4	3
PP	Permissions Policy	Working Draft	W3C	February 2023	1	1
UIR	Upgrade Insecure Requests	Editor's Draft	W3C	October 2022	2	2
Fetch	Fetch	Living Standard	WHATWG	March 2023	8	7
HTML	HTML	Living Standard	WHATWG	March 2023	3	3

Table 1: Considered specification documents and extracted rules

While some previous works tried to use semi-automated techniques using natural language processing [65], the results of current NLP technology are still unsatisfactory for texts as complex as the HTTP specifications, and we instead manually parsed the specification documents. Guided by the RFC keywords, we scanned the documents for sentences specifying what rules participants of the HTTP protocol have to follow. If a rule matched our above-defined criteria, we saved it in a database, including a name, short description, verbatim text from the specification document, link to the document, and several other attributes explained below.

We attach a specification *level* to each rule. The level can be Requirement (MUST in specification if RFC), Recommendation (SHOULD in specification if RFC), or ABNF (Augmented BNF [20]; defines the grammar for HTTP fields).

ABNFs are usually required to be followed by senders; for receivers, more lax parsing rules are often defined as legacy clients are known to send values that do not conform to the specified syntax. Contrary to their name, recommendations in RFCs are not only an implementation suggestion but are by default to adhere to: "there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course." [12].

In addition, we attach a test *type* to each rule. Types explain how a rule can be violated. We distinguish two main groups: single and multi. *Single* means the test property can be violated by a single response, potentially only if the response is to a specific probing request, such as a particular request method. *Multi* means the test property can only be broken by several related responses, e.g., HEAD responses should be similar to GET responses for the same resource.

3.2 Testing Framework

All considered rules specify invariants of request-response groups. Thus, we must send requests to test targets and analyze their responses to test HTTP conformance. We create one test for each rule, and each test analyzes one-to-many request-response pairs.

We created each test case as a Python function and tested them by deploying a violating and a non-violating endpoint using WPTserve [85]. We could not host a violating endpoint for a small number of test cases as the underlying HTTP stack of WPTserve using Python did not allow us to send some severely broken responses even though the tool's purpose is sending invalid HTTP responses.

The single test rules are split into *probe* and *direct* tests. For *probe* test rules, we generate a large number of syntactically valid probe requests. By using an array of various probe requests, we aim to induce a wide range of behavior in HTTP implementations. In total, we send 180 probe requests (HTTP/1.1, HTTPS/1.1, HTTPS/2, several HTTP methods, and several headers). We send these requests with a standard HTTP client (Python HTTPX [23]). We run each test case on all responses received for all probe requests. For *direct* test rules, we send syntactically invalid requests, e.g., a request without a Host Header. Such invalid requests are not allowed by most HTTP client tools. We thus send them over sockets directly, with and without HTTPS, using HTTP/1.

For the *multi* test rules, we give each test case access to all requests and responses obtained for a URL. The test case then uses all responses necessary, e.g., all pairs of responses belonging to requests that either use HEAD or GET but are otherwise identical.

Given a URL to be tested, we run all probe requests on it. The HTTP client is connected to a proxy (Mitmproxy [19]), and every *probe* test is automatically run on each received response. The probe requests are rate-limited, and we use one proxy per tested origin. After all probe requests have been run, we invoke each *multi* test with all request-response pairs as input. Lastly, we run all *direct* tests on the URL. Two direct tests are only run once per host and not per URL as they are only applicable to the server as a whole and do not specify a URL, e.g., CONNECT-related tests. We do not run *direct* tests on real websites due to ethical concerns, as explained in Section 6.2.

We save each request and response pair in a database, including the corresponding headers and bodies, for later analysis. In addition, we save the test outcome for each applicable request/response pair.

4 EVALUATION

For our evaluation, we crafted tests for a total of 106 extracted rules as shown in Table 2. Looking at the specification level of each rule, we see that requirements (41) are the most prevalent, followed by ABNFs (38) and recommendations (27). We implemented most of them as *probe* tests (89), followed by *direct* tests (11) and *multi* tests (6).

Specification Level	Test Type Probe	Direct	Multi	All
Requirement	29	8	4	41
ABNF	38	-	-	38
Recommendation	22	3	2	27
All	89	11	6	106

Table 2: Overview of the considered HTTP rules

We test two groups of HTTP implementations: popular web servers and real websites. For popular web servers, we used servers from the w3techs report [77]. For real websites, we used the CrUX dataset of February 2023 and tested the Top 5,000 origins, and additionally, 5,000 randomly sampled origins ranked between the Top 500K and 1M. The list of tested hosts and Dockerfiles for the local servers are available online [60]. The local server versions are listed in Table 6 in the appendix.

4.1 Local Servers

Out of the ten most popular web servers according to w3techs, we test five (Nginx, Apache, OpenLiteSpeed for Litespeed, Node.js, and Caddy). The other five are either not available for self-hosting (Cloudflare Server, Google Servers, IdeaWebServer), only work on Windows (Microsoft-IIS), or do not work standalone as they require an upstream server (Envoy). In addition, we test three more popular web servers (Jetty, Tomcat, OpenResty) and one reverse proxy that can run without configuring any upstream servers (Traefik). We left most of the configurations of the nine tested servers in their *defaults*. However, we enabled both HTTPS and HTTP/2 for all but Node. We configured all servers to host a simple HTML file. Traefik cannot host static files, and we did not configure any upstreams; thus, it usually returns a *404 response*.

For each tested server apart from Node, we test both the HTTP and HTTPS versions of the server. We visit the *landing page* (/) and a URL that should not exist (/{32randomalphanumchars}). For every URL, we first run all 180 probe requests, then the 13 direct tests, and lastly, we run the multi tests on the collected responses. In total, we perform 6,120 probe requests and 816 socket requests for the nine servers.

Local Server Results: Table 3 shows the results for the local servers. 16 rules were violated at least once, and every server violated at least six rules (maximum ten). Two test cases were violated by all nine servers, and four test cases were violated only by one server. Seven of the broken tests were direct, six probe, and three multi tests. We discuss security implications in more detail in Section 5.

4.2 Popular and Long-Tail Hosts

The tested local servers only use a limited feature set of HTTP as we only configured them to host a simple static file. Thus, we also tested real websites in addition to the locally installed web servers. Most of these sites do not host static files but instead provide rich applications. Also, their requests and responses pass through plenty of middleboxes, such as caches and load balancers, which can also break the rules and lead to more complex and dangerous behavior.

Test Type	Spec Level	Rule Name	apache	caddy	jetty	nginx	node	openlitespeed	openresty	tomcat	traefik	All
Direct	Req	Bad host	●	●	●	●	●	●	●	●	●	9
Probe	Rec	Redirect after HTTP-UIR	●	●	●	●	●	●	●	●	●	9
		STS after UIR*	●	●	●	●	○	●	●	●	●	8
Direct	Rec	Blocked methods result in 405	●	●	●	○	●	●	○	●	●	7
		Unknown methods result in 501	○	●	●	●	●	●	●	○	●	7
	Req	Illegal chars	○	●	●	●	○	●	●	●	●	7
Probe	Req	Allow header present for 405	●	○	○	●	○	○	●	○	○	3
		Forbidden status-codes for POST	●	○	○	○	○	●	○	●	○	3
Direct	Rec	Allow CRLF prior to request line	○	●	○	○	○	○	○	○	●	2
	Req	Illegal whitespace after startline	○	○	○	○	●	●	○	○	○	2
		Illegal whitespace in header name	○	○	○	○	●	●	○	○	○	2
Probe	Req	Date header required	○	○	●	○	●	○	○	○	○	2
	ABNF	Server ABNF	○	○	●	○	○	○	○	○	○	1
Multi	Rec	HEAD and GET same headers	○	○	○	○	○	●	○	○	○	1
	Req	Content-Length for HEAD=GET	○	○	○	○	●	○	○	○	○	1
		Same headers for 304 and 200	●	○	○	○	○	○	○	○	○	1
		All	7	7	8	6	8	10	6	6	7	65

Table 3: Violating servers for each broken rule (● violating, ○ non-violating; highlighted rows are explained in Section 5; * only applies to STS-safe hosts)

We took the top 5,000 origins from the February 2023 CrUX dataset [22] and 5,000 origins from the Top1M bucket (ranked between 500k and 1M). We took the hostname of every origin in this dataset and tested each host with HTTP and HTTPS. As CrUX includes HTTP and HTTPS origins separately, hosts can occur two times. In our dataset, ten hosts occurred twice. Thus, we tested a total of 9,990 distinct hosts and 19,980 origins. In addition to the landing page and the non-existing page, if the request to the landing page is redirected to a same-origin URL, we also test that URL to test a URL that is likely to neither be redirecting nor cause an error such as *404 Not Found*. If an initial standard HTTPX GET request to the landing page throws an error or times out, we mark the origin as non-reachable. Before we test each URL, we perform one standard HTTPX GET request to the URL under test. If this request does not throw any error, we start running all our probe requests against it; otherwise, we abort running tests for this URL.

To not accidentally cause any havoc, we do not run our direct tests against real websites, as, e.g., syntactically invalid requests might cause issues such as HTTP Request Smuggling, which we discuss later. Additionally, we refrain from potentially dangerous probes using DELETE. In total, we ran 6,003,200 probe requests, out of which 27,417 (0.46%) failed according to our HTTPX test runner.

The results for all violated rules in the wild are presented in Table 5 in the appendix. The most broken rule applies to 8,440 (89.14%) hosts, whereas many rules only are broken on a small number or even only one host. A total of 9,431 (99.61%) hosts broke at least one test case, and 68 test cases got broken at least once. The maximum number of broken tests for one host is 16.

Comparison of popular and long-tail hosts: The results for popular and less popular hosts are similar, and we could not discover any influence of the rank on HTTP conformance. The average number of violations per host is 4.69 for both groups. In addition, 60 unique rules are broken across all popular hosts and 59 for the long tail.

5 SECURITY IMPACT

As explained in Section 2.4, we want to highlight that even without an existing exploit, every inconsistency is a risk. In general, every broken HTTP rule violation and inconsistency between HTTP processors is bad as they can lead to various issues such as interoperability and functionality loss, non-optimal caching, negatively influencing web measurement tools or search engine robots, or increasing the attack surface due to inconsistent security header behavior in browsers, or semantic gap attacks.

Still, the potential negative impact of a violated rule highly depends on both the rule and how exactly it is broken. In the following, we discuss three groups of potentially hazardous rule violations (everything in Table 4 and the rows shaded in Table 3) and additional problematic issues we discovered while running our tests. To decide whether a rule could be considered dangerous, two authors went through the list of rules and, for each rule, discussed whether it is related to security. In total, we classified 55 rules as security-relevant, out of which 37 were broken at least once.

5.1 HTTP(S) Issues

HTTP is inherently prone to MITM attacks, and HTTPS, *Strict-Transport-Security (STS)* and *Upgrade-Insecure-Requests (UIR)* can be used to mitigate such attacks. If the *UIR* header is received in an HTTPS request, a server that supports *STS* should send a Strict-Transport-Security header. If the UIR header is received in an HTTP request, a server should redirect to *trustworthy URL*, i.e., to HTTPS [75, sec.3.2.1]. The two rules STS after UIR and Redirect after HTTP-UIR test for this behavior, and our results show that most hosts do not deliver the best possible security.

The first rule only applies to STS-safe hosts. We cannot know whether a host is STS-safe; however, out of the 8,440 violating hosts, 2,386 hosts served an STS header at least once and are thus clearly not following the specification. Additionally, given the widespread adoption of free Let's Encrypt certificates [13, 24, 25], it is best practice to be STS-safe, i.e., sites should be reachable through HTTPS. The second rule applies to all HTTP hosts; however, many hosts performed no redirect for certain requests but instead returned status codes such as 405. From all the 9,468 reachable hosts, 252 were only reachable by HTTP. However, even they could follow the specification by redirecting to a URL with a different host. Violations of both these rules could lead to unsafe HTTP connections and should thus be avoided.

5.2 Security-Related Headers

Duplicate, missing, or invalid headers can have negative consequences for security. In the following, we first describe potential consequences for six groups and finish with an analysis of reasons for rule violations concerning headers.

5.2.1 Cookies. Even though cookies were never meant for security, they are nowadays a cornerstone for critical security functionality such as authentication and authorization. While cookies with the wrong date format might only have an unintended lifetime, inconsistencies in the interpretation of a set-cookie string as well as duplicate cookies could break the confidentiality and integrity of cookies even in the presence of countermeasures such as *__HOST*

Group	Rule Name	#Hosts
HTTP(S) Issues:		
	STS after UIR*	8,440
	Redirect after HTTP-UIR	7,822
Security Related Headers:		
Cookies	Cookies use IMF-fixdate	2,850
	Duplicate cookie names	518
	Set-Cookie ABNF	93
	Cookies with duplicate attributes	22
STS	STS not allowed for HTTP	550
	STS ABNF	15
	Duplicate directives for STS	1
Duplicates	Duplicate headers	465
MIME	Content-Type header required	290
	Content-Type ABNF	48
	XCTO ABNF	10
Restrictive	XFO ABNF	262
	Duplicate CSP	71
	CSP ABNF	23
	PermissionsPolicy ABNF	18
	Duplicate CSP-RO	7
	COOP ABNF	6
	CORP ABNF	3
	COEP ABNF	1
CORS	AC-Allow-Origin ABNF	83
	AC-Allow-Credentials ABNF	28
	AC-Allow-Methods ABNF	19
	AC-Allow-Headers ABNF	18
	AC-Max-Age ABNF	3
HRS Primitives:		
	Forbidden Content-Length for 1XX and 204	55
	Forbidden surrounding whitespace for fields	51
	Forbidden content for 304	46
	Content-Length ABNF	3
	Upgrade required for 101	1
	TE forbidden for non HTTP/1.1 responses	1
	TE forbidden for 1XX and 204	1

Table 4: Potentially dangerous violated rules in the wild (* only applies to STS-safe hosts)

prefixes [68]. The most prominent violation was the incorrect usage of dates. The specification requires the usage of IMF-fixdates [6, sec.4.1.1], however 2,850 sites used other date formats and, for example, used dashes instead of spaces to separate day, month, and year.

Such dates may lead to different browsers interpreting cookies differently, which in the worst case, could lead to cookies not being deleted, as this is usually achieved by setting an expiry date in the past. Second, 518 hosts delivered cookies with duplicate names, which should not be done according to the specification [6, sec.4.1.1]. Duplicate cookies may cause different browsers to rely on either the first or last occurrence of the cookie, which may cause inconsistencies. Moreover, 93 hosts violated the ABNF for Set-Cookie. As shown by Squarcina et al. [68], the error tolerance in attempting to parse cookies that fail to follow proper syntax leads to several

security problems. However, strict parsing would ignore cookies for the affected hosts, which causes functionality issues.

5.2.2 STS. *Strict-Transport-Security* can be used to enforce HTTPS connections. As shown in Section 5.1, most websites do not deliver STS headers consistently even if requested with *Upgrade-Insecure-Requests*. For those hosts that sent an STS header, we conducted further analyses. Importantly, 550 hosts sent the STS header through an insecure HTTP connection. Browsers ignore the STS header in HTTP connections to ensure no breakage in functionality for misconfigured HTTP origins. The correct way to implement STS is to redirect the client to the corresponding HTTPS origin and set the STS header there. The prevalence of the misconfiguration highlights potential misunderstanding by operators, which may well believe that setting the STS header saves them from network attackers, yet their configuration has no impact on security whatsoever. Similarly, broken ABNFs (15 hosts) and duplicate directives also render the STS header invalid, nullifying its seeming protection.

5.2.3 Duplicate Headers. Many headers are not allowed to occur several times in a response. On 465 hosts, at least one of the following headers occurred twice: strict-transport-security, x-frame-options, x-content-type-options, content-type, retry-after, server, access-control-allow-origin, expires, age, report-to. Depending on the header, a browser might choose the first or last or use some other complex processing procedure. In either case, the resulting behavior might not be the most secure choice, nor what the developer wanted, and behavior might differ between browsers. As shown by Roth et al. [62], such issues may even lead to entirely undermining well-designed protection, such as the accidental introduction of `max-age=0` for STS, which disables the mechanism altogether if contained in the *first* header observed by the client.

5.2.4 MIME Sniffing. To inform browsers of the resource type and how they should handle it, servers need to send the *Content-Type* header. This header is paramount since otherwise browsers are forced to use so-called MIME sniffing. MIME sniffing is the process of scanning the first bytes of the response to deduce the content type. This process, however, has been shown to lead to issues like XSS if uploaded images are misinterpreted as HTML or script content [82, 7]. If no content-type header is specified at all (Content-Type header required), if the specified content-type is invalid (Content-Type ABNF), or if MIME sniffing is not correctly disabled (XCTO ABNF), the potential for MIME sniffing issues is high. As our tests showed, 290 hosts failed to deliver a content type header at least for one request, even with our unintrusive tests. These results hint that we could only cover the tip of the iceberg, and such issues may be more prevalent in practice. Moreover, 48 hosts had invalid values for their content types, and 10 sent a malformed XCTO header.

5.2.5 Restrictive Headers. The headers in the *restrictive* group all instruct browsers to (not) allow some security-related behavior. Invalid values can lead to inconsistent behavior between browsers [15, 83, 66] and only some users being protected. Additionally, it can lead to a false sense of security (*If a security header is set, my site will be secure*). However, if the header is invalid, the security feature will often not be activated, and the browser will fall back to an insecure default. Due to malformed and inconsistent HTTP, browsers often introduce error tolerance, and parsing gets more complex over time. Complex error tolerance can introduce various issues that either only occur in some browsers or regresses between versions [15, 83, 33, 49]. Notably, the majority of cases (XFO with 262) were caused by the specification being updated years after the header was first deployed. Moreover, the Duplicate CSP case highlights an issue between specification and practice: composing two CSPs is the gold standard for security [79, 78]. Even though a CSP can be composed in two ways, by sending a list in one header (using a comma) or by sending several CSP headers, the specification recommends not to use the second way [74, sec.3.1].

5.2.6 CORS. Lastly, even for features with secure defaults such as CORS, incorrect header values are problematic as they can lead to developer frustrations if they do not get it to work and often lead to overly unrestrictive or insecure values such as origin reflection or allowing everything (∗).

5.2.7 Reasons and Examples. We have identified six common reasons for how and why these rules were broken. First, many ABNFs were broken due to the incorrect separation of several elements within a header or between a value and a parameter. Depending on the header, this has to be done with commas, semicolons, spaces, or equal signs, confusing developers. Second, some violating headers partly used the syntax from another header or seemed to be mistaken for another header entirely. For example, we observed *X-Frame-Options* headers with CSP-style syntax, such as *None* or *, and we detected permission policy headers using the old feature policy syntax. Third, we observed many values that were either allowed in the past (e.g., XFO: allow-from) and are now deprecated or never existed but, from a common-sense viewpoint, might exist (e.g., XFO: allowall). Fourth, many responses appeared to entail typos in directive names and similar. For example, *Thu, 11 May 2023 07:0309 GMT* seems to be a manually formatted date where the second colon was forgotten. Fifth, we also discovered invalid headers related to incorrect encoding and failed templating. Some of these should have been several headers but were delivered as one header by the server, as linebreaks were not interpreted correctly. One example is: *Connection: Close
Content-Type:text/html*. Other headers had a literal value of *undefined* or contained strings such as *%{HTTP_HOST}*, clearly indicating failed templating. Finally, many responses incorrectly specified headers multiple times or used a list format, whereas only a single value is allowed. We observed both responses with the same value repeated and with differing values. We speculate that headers such as X-Frame-Options are often set both by an origin server and by another entity in the HTTP processing chain, such as a reverse proxy.

5.3 HRS Primitives and Host of Troubles

The most famous attacks related to HTTP parsing are HTTP Request Smuggling (HRS) attacks [37, 42]. These attacks are usually related to varying interpretations of the limits of messages by participating protocol partners such as a reverse proxy and an origin server. Such parsing discrepancies are often caused by multiple or invalid *content-length* and *transfer-encoding* (TE) headers or other assumed invariants related to the body of HTTP messages. Although the basic examples such as two *content-length* headers are

fixed in all major server and proxy implementations, new, more complex issues are discovered every year, for example, related to HTTP/2 to HTTP/1.1 conversion [41].

In the following, we present violated rules that we classified as HRS primitives in the wild and local servers. Such violations do not necessarily have to lead to HRS but pose a risk to existing and new HTTP processors in a message chain if not handled properly and consistently.

5.3.1 HRS Primitives in the Wild. For some status codes, HTTP versions, or responses to certain methods Content-Length or TE headers are not allowed, and their presence could confuse HTTP tools that do not validate this requirement (Forbidden Content-Length for 1XX and 204; TE forbidden for non HTTP/1.1 responses; TE forbidden for 1XX and 204). In addition, incorrect content-lengths (Content-Length ABNF) or transfer-encodings can also lead to differences in parsing. Also, responses with status code 304 are not allowed to have any content (Forbidden content for 304). We discovered 46 hosts that send bodies in HTTP/2 responses with status code 304. If a reverse proxy does not reject the body of such a response, it could interpret the erroneous body as the following message and go out of sync with the origin server. Similar things have occurred in the past when a HEAD response incorrectly contained a body [21].

Secondly, incorrect whitespace in header names or values can confuse HTTP processing chains as some entities strip them, whereas others ignore the entire line or process them as is. Forbidden surrounding whitespace for fields denotes behavior where whitespace was encountered around field values that could lead some processors to ignore the field.

Thirdly, the Upgrade required for 101 rule states that an upgrade header is required for responses with status code 101. If this header is missing, one cannot know which protocol it is being switched to, and it can also lead to HRS [1].

5.3.2 Local Servers. The following violations concern the correct handling of invalid requests and could thus only be tested for the local server installations.

A prominent issue is the differing handling of invalid characters. The specifications state that "Field values containing CR, LF, or NUL characters are invalid and dangerous, due to the varying ways that implementations might parse and interpret those characters" [28]. Our Illegal chars test revealed that 7/9 servers did not return status code 400 for at least one of the forbidden characters. OpenLiteSpeed allows \00, \r, and \n. Caddy, Traefik, Tomcat, and Jetty allow \n. Nginx and Openresty allow both \r and \n. Apache and Node correctly reject all three invalid requests.

Another issue is whitespace in header names. The rules Illegal whitespace in header name and Illegal whitespace after startline show that Node and OpenLiteSpeed do not reject requests with illegal whitespace properly.

Lastly, while not an HRS primitive per se, the Bad host rule is also critical as differing interpretations of the host header can lead to host of troubles attacks [17]. The rule requires a request with a bad host header to be rejected with status code 400. We used no host, an invalid host (ABC), and two host headers, and each tested server did not return status code 400 for at least one of the three invalid options. Apache returns status code 200 for an invalid host but correctly rejects requests with two hosts or no hosts. OpenLiteSpeed and Node allow all three incorrect requests. Jetty always returns code 400 for HTTPS but allows an invalid host for HTTP. Caddy redirects invalid hosts for HTTP and allows them for HTTPS.

These examples show that popular web servers violate core HTTP rules and behave inconsistently, even in default configurations.

5.4 Additional Problematic HTTP Behavior

In addition to the violated rules, we also observed other non-HTTP-best practice behavior not covered by our tests in the wild. 106,127 HTTP/1.1 probe requests received an HTTP/1.0 response. 1,349 probe requests received responses with invalid body lengths (usually expected 0 bytes, received a couple hundred bytes). 3,088 responses were unexpected by the proxy either because they were HTTP/0.9 responses or out of order. 100 responses contained spaces between a header name and the colon. 548 responses failed to decompress, and several other errors occurred, such as 10 switch upgrade events without a pending request or 71 responses with invalid chunk headers. These results indicate that a few deployed HTTP systems do not even conform to the most basic requirements.

6 DISCUSSION

In this section, we first present the limitations and ethical considerations of our work. After that, we carefully analyze our insights and suggest ways to improve the dismal HTTP conformance situation in order to arrive at a more secure web.

6.1 Limitations

Our 106 extracted HTTP rules do not cover all of HTTP. Rules regarding HTTP are not only defined in the 12 documents we considered for this work but certain features are defined in other documents. In particular, we did not consider HTTP/3 (RFC9114) [11], as the HTTP proxy and client we used in our experiments did not support it. Also, our manual rule extraction process is not free from subjectivity, we might have missed or misinterpreted rules. In addition, we had to exclude many Requirements and Recommendations as they were not black-box falsifiable, too vague, or only applied to specific protocol participants such as caches. Finally, many parts of the specifications are optional, or choices exist, resulting in various implementation differences, we could not discover with our framework.

Our tests are thus inherently incomplete, and we cannot show that a website or server follows *all* HTTP specifications. Furthermore, we cannot conclusively say that a website is more conformant than another only because it breaks fewer rules in our tests than another, as rules might be of varying importance, and other, non-tested rules might be broken too. However, our tests show that HTTP conformance is dismal and that many websites break various clearly defined rules.

Moreover, we cannot reliably attribute a violated rule to one specific entity in a complex HTTP processing chain, as any intermediaries might generate or modify the response. However, as all our rules apply to general responses, this means that the HTTP configuration of the site, including its intermediaries, is not HTTP conformant.

Finally, we only tested three URLs for nine servers and 9,990 hosts, for which we only ran a subset of tests due to ethical considerations. More rule-breaking behavior might be discoverable by testing more URLs or testing in a different state, such as login. Also, depending on geolocation or crawl time, one might receive different responses [62, 43].

6.2 Ethical Considerations

The most problematic issues we discovered are well-intentioned, yet incorrectly configured security headers (e.g., an invalid STS header voids any protection) and the HRS primitives. We notified operators through email to *webmaster@domain* for these issues. Testing for HTTP conformance requires sending potentially dubious requests to websites and analyzing the responses. Such a process yields obvious ethical concerns as our requests might interfere with the regular operation of the websites. To minimize potential harm, we only send syntactically valid requests to websites using a standard HTTP client (Python HTTPX [23]). We did not send DELETE requests to websites to not accidentally delete any resources on the servers. Also, the *direct* tests that require sending malformed requests are only used against locally hosted web servers and not against real websites, as they might lead to denial of service or HTTP request smuggling.

In addition, we rate-limited our crawlers to avoid overwhelming websites with requests. We only send one parallel request per origin and wait two seconds between each probe request. Further, we followed best practices and used a custom user-agent header leading to a website explaining our experiment with the option to opt out of our experiments. Finally, we add *cache-control: no-store* to each request to ensure responses to our requests are not saved in any intermediary caches to avoid Cache-Poisoned DoS attacks [52]. With all these safeguards installed, we believe that the potential benefits of our work discovering issues in HTTP conformance of websites outweigh potential negative consequences.

6.3 Insights and Problematic Consequences

One of the most impactful vulnerabilities in Web security in recent years was the widespread finding of HTTP Request Smuggling [44]. One of the most significant contributors was the fact that some servers would inconsistently process requests with both Transfer-Encoding (TE) and Content-Length (CL) headers, even though the specification explicitly says to ignore CL if TE is set to *chunked*. These vulnerabilities show how hazardous it is if interpretations of the HTTP specification diverge.

Currently, both implementations and the specifications often allow certain malformed responses and introduce many special rules leading to complex and hard-to-understand specifications and error-prone code instead of strictly rejecting invalid messages. One solution would be to change this behavior. However, many current systems do not follow the specifications, and an implementation that rejects all invalid responses would be unable to interact with a large part of the web, posing a difficult trade-off. For example, Cloudflare decided to accept a wide array of invalid HTTP messages for compatibility reasons [86].

In addition, the fact that there is no single specification of HTTP is problematic. As rules are defined in isolation, they can interfere with each other, creating unnecessary tension. For example, Redirect after HTTP-UIR specifies that a server should redirect HTTP requests with a UIR header to an HTTPS URL; at the same time code 405 blocked methods specifies that if a blocked method is encountered, a response with status code 405 should be generated. As responses with status code 405 do not redirect, it is impossible to fulfill both rules simultaneously if a request with a blocked method and a UIR header is encountered. Additionally, specifications sometimes discourage secure behavior. For example, best practice for CSP stipulates that two policies should be deployed [79]. However, the specification [74] notes that having two separate headers is bad. The correct (by the definition of the specification) solution is to use a comma-separated list of CSPs in *one* header. Given the rules for folding headers, this is *de facto* the same as sending two headers. Even though setting and updating two headers is likely less complex and error-prone, operators choosing two separate headers then act against the specification[1].

Since the early days of the web, error tolerance has been a vital issue. In the first "browser wars" [87], browsers were eager to steal customers away from their competitors by rendering anything that even remotely resembled HTML. The effects of this are still felt today through attacks like mutation-based XSS or dangling markup [39, 38]. Similarly, browsers ignore invalid headers without notifying users of the (lacking) protection. In the best case, warnings are shown in the browser console. However, in the case of XFO, these warnings only are shown when loading the page in an iframe. That is, a developer visiting their own site directly is unable to recognize the issue. In contrast, for HTTPS deployment, browsers went from warning users of insecure origins when entering passwords [25] to no longer showing a positive green indicator on secure connections today, but instead explicitly marking insecure connections with a negative indicator (stating *Not Secure* next to the URL).

6.4 Towards Better HTTP Conformance

In the following, we outline how to improve HTTP conformance in the wild based on our findings.

6.4.1 Lack of Specification Testing. We believe that one of the core reasons for the current state of affairs is that no common test infrastructure or reference implementation for HTTP exists, and implementers are thus left alone to read the specification or implement whatever they think works. The current specifications are hard to understand and are distributed over many documents that are not necessarily all updated simultaneously and might contradict each other.

In the area of browsers, the situation was similar until recently when the community and browser-driven efforts for a shared test platform resulted in *web-platform-test* [84] in 2018. Although problematic browser differences are still regularly discovered, e.g., related to CSP [83] or XS-Leaks [46, 59], the WPT platform helps browsers and developers fix such issues in a cross-browser compatible way. In the area of HTTP, nothing comparable exists. For intermediaries such as proxies, the CoAdvisor project existed in the past. However, the project was stopped due to a lack of interest, and

[1]We reported this issue to W3C and the recommendation was removed: https://github.com/w3c/webappsec-csp/pull/622

the last archived version only supports RFC 2616 [70]. For general HTTP responses, the HTTP linter REDbot [56] exists. However, this project does not support the new RFC 9110 specifications and misses many functionalities, such as HTTP/2 support. The interest in the project also appears low, and we could find, report, and fix a bug in a header ABNF check that has existed for six years[2].

We open-source our test suite as a starting point for such a testing effort [60]. We hope that it sparks interest with server developers and the web community so that many more test cases can be added in the future. For many rules, where the specifications are vague, a consensus on the correct behavior could be formed. Moreover, we suggest providing better documentation and an overview of the existing specifications to help implementers find the relevant information. Within the specifications, failure behavior should be made more explicit rather than allowing implementers to choose how they handle invalid requests or responses. We also suggest reducing the number of optional features and the number of possible choices as they result in diverging behavior. Moreover, we suggest considering our results and continuously measuring the web for conformance, and reconsidering recommendations and requirements that are frequently broken to see whether they (still) make sense. Finally, more examples for both valid and invalid cases in the specifications could help developers.

6.4.2 Reconsidering the Robustness Principle. An underlying cause for many of the discovered issues is the robustness principle [57, 58, 16] also known as Postel's law that states "be conservative in what you do, be liberal in what you accept from others." [57]. Due to how many implementations apply and (mis)interpret the principle on the web nowadays, most participants of the HTTP protocol do not strictly reject invalid requests or responses but silently try to repair them or ignore only the invalid parts.

Thus, sending an invalid request or response usually does not cause any immediate breakage but instead seems to work. The sender can, therefore, not directly realize their mistake. Moreover, specifications often contain different rules for sending and receiving (e.g., of a specific header). As a result, the specifications become complicated, and many senders choose to send messages allowed by the receiving specification even though it is not allowed by the sending specification. As a final consequence, this can lead to many hard-to-detect and fix issues such as HTTP Request Smuggling [37] or Host Of Troubles attacks [17]. Such issues are hard to debug, as it is unclear what every HTTP processor in a processing chain does.

The principle or the currently observable interpretation of the principle has been criticized repeatedly [5, 64, 17, 73]. We also note this to be one of the key factors to the issues we discovered in the wild and call on standardization bodies and vendors alike to ensure that (a) the same rules apply to senders and receivers and (b) any requests/responses which violate a rule must be rejected.

6.4.3 More User-Facing Warnings. On the modern web, almost all websites are reachable through HTTPS. The recent success of HTTPS is not only because of the availability of free CAs like Let's Encrypt but also because browser vendors incentivize operators. They gradually went from having a positive green indicator for secure connections through having a red indicator on insecure ones all the way to only showing the negative red indicator for any site without HTTPS. Additionally, Google incorporates the usage of HTTPS into their ranking [36], which effectively forces most sites to enable secure connections.

In line with this, browsers should also be more explicit about website misconfigurations. In that scenario, both users and operators would be informed about mistakes in the sites they visit or run, which would incentivize fixes to avoid a bad reputation.

[2]https://github.com/mnot/redbot/pull/306

7 RELATED WORK

This section presents other works focusing on conformance testing efforts in HTTP, the worst possible consequences of non-conformity: semantic gap attacks, research that revealed non-conformity in the past, and conformance testing in browsers and other protocols.

HTTP specification analysis and conformance. Shortly after the initial draft standard specification of HTTP/1.1 in RFC 2616 in 1999 [55], Krishnamurthy et al. investigated compliance of popular websites by testing for essential functionality such as support for the HEAD method or for persistent connections showing that many sites are not compliant [48]. In 2008, Adamczyk et al. performed a similar study, only focusing on the methods GET, HEAD, OPTIONS, TRACE, and CONNECT, showing that many sites are still not compliant [3]. Similarly, The Measurement Factory started an industry project for testing HTTP compliance of intermediaries in 2001 [71]. However, this project was discontinued due to a lack of demand [70]. The area of caching in HTTP, including conformance to the specifications, was studied extensively by Nguyen et al. [51, 53, 52]. Since 2009, REDbot [56], the linter for HTTP, can check for HTTP issues on websites. However, it is only used as a check-yourself application rather than to study the general status of the web, has a limited amount of supported test cases, and still needs to be updated to the newest specifications.

In contrast to the first two studies, rather than testing for basic features, we verify that no requirements or recommendations are violated in received responses. In contrast to CoAdvisor, our tests focus on responses received after an arbitrary HTTP processing chain and not testing intermediaries in isolation. REDbot is the most similar to our work, and we based our ABNF implementations on it. However, it has never been used in a measurement study and has no focus on the security impact of non-conformity.

Semantic gap attacks. In an HTTP processing chain, usually, several different processors are involved. If they differ in their semantic interpretation of the messages, this can have dramatic consequences called semantic gap attacks. Several works studied individual instances of such attacks, such as HTTP Request Smuggling [42, 41], Web Cache Deception and Cache Poisoning [50], and Host-Of-Trouble [17].

We took a step back from concrete attacks and studied general HTTP conformance, as violated HTTP rules are often a starting point for these attacks.

Web (security) measurements. Many studies measuring the web regularly show inconsistent and incorrect behavior by many sites. While usually, it is not their primary goal, they provide evidence

that many aspects of the HTTP specification are violated. For example, many studies have shown that not only do most websites configure an insecure content security policy, many sites even deliver malformed policies breaking their ABNF [79, 61]. Other works showed forbidden duplicate headers or syntax errors in XFO and STS headers [15, 66]. In addition, several projects such as Chrome Platform Status [35] or Web Tech Survey [69] collect statistics about web traffic. However, these collect data mostly from benign, non-ambiguous requests and responses, mainly counting the correct feature usage and not violating instances.

Browser compatibility and conformance. Many works have shown issues in browser implementations [83, 34, 45] or between browser implementations [49, 40]. These include security issues, such as a CSP not being applied in a browser, and functionality and interoperability issues. To be as interoperable as possible, browser vendors started the web platform test project, where they try to implement features correctly and consistently between browsers and specifications, including HTTP-related features [84].

Other protocols. The fact that implementations not following the specification or applying differing interpretations lead to issues is well-known in TCP [47], URL parsing [67, 4], or HTML [38]. To help developers, the W3C provides an HTML and CSS validator as well as a URL testing suite [76].

8 CONCLUSION

Consistent interpretation of specifications is essential for protocols with several communication partners. Different interpretations or intentional misbehavior can lead to issues such as DoS, broken functionality, or semantic gap attacks. In this work, we studied the landscape of HTTP conformance of widespread web server implementations and real-world websites.

To that end, we extracted 106 falsifiable rules from HTTP specification documents and created over 100 test cases. By running this test suite against 9,990 real hosts and nine local servers, we could show that HTTP conformance is grim, with most tested implementations violating at least one rule and more than half of all rules violated at least once, including security issues such as HRS primitives, MIME sniffing, and insecurely used security headers. For example, 8,440 hosts failed to deploy STS after upgrading insecure requests, and 550 hosts specified STS through HTTP, indicating a lack of understanding of how the header works. Given that browsers to not inform the developer about the misplaced header through the console, such issues often remain hidden.

Our findings suggest varied causes, such as complicated and vague specifications, missing direct negative feedback, and a lack of testing infrastructure. By open-sourcing our test suite, we hope to initiate a shift towards a more HTTP-conformant and thus secure web.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. In addition, we thank our student helper Eduard Ebert for his assistance during a prestudy.

This work was conducted in the scope of a dissertation at the Saarbrücken Graduate School of Computer Science.

AVAILABILITY

We open-sourced our tool chain and it is available at https://github.com/cispa/http-conformance.

REFERENCES

[1] [SW] 0ang3el, 0ang3el/Websocket-Smuggle 2019. URL: https://github.com/0ang3el/websocket-smuggle.
[2] Erwan Abgrall, Yves Le Traon, Martin Monperrus, Sylvain Gombault, Mario Heiderich, and Alain Ribault. 2012. XSS-FP: Browser Fingerprinting using HTML Parser Quirks. arXiv: 1211.4812 [cs]. preprint.
[3] Paul Adamczyk, Munawar Hafiz, and Ralph E. Johnson. 2008. Non-compliant and Proud: A Case Study of HTTP Compliance. https://hdl.handle.net/2142/11424.
[4] Dashmeet Kaur Ajmani, Igibek Koishybayev, and Alexandros Kapravelos. 2022. yoU aRe a Liar://A Unified Framework for Cross-Testing URL Parsers. In *IEEE Security and Privacy Workshops*. SecWeb. DOI: 10.1109/spw54247.2022.9833883.
[5] Eric Allman. 2011. The Robustness Principle Reconsidered. *ACM Queue*. DOI: 10.1145/1989748.1999945.
[6] Adam Barth. 2011. HTTP State Management Mechanism. Request for Comments RFC 6265. Internet Engineering Task Force. DOI: 10.17487/RFC6265.
[7] Adam Barth, Juan Caballero, and Dawn Song. 2009. Secure Content Sniffing for Web Browsers, or How to Stop Papers from Reviewing Themselves. In *IEEE Symposium on Security and Privacy*. DOI: 10.1109/sp.2009.3.
[8] Mike Belshe, Roberto Peon, and Martin Thomson. 2015. Hypertext Transfer Protocol Version 2 (HTTP/2). Request for Comments RFC 7540. Internet Engineering Task Force. DOI: 10.17487/RFC7540.
[9] Tim Berners-Lee. 1991. HTTP 0.9. https://www.w3.org/Protocols/HTTP/AsImplemented.html.
[10] Timothy J Berners-Lee. 1989. Information Management: A Proposal.
[11] Mike Bishop. 2022. HTTP/3. Request for Comments RFC 9114. Internet Engineering Task Force. DOI: 10.17487/RFC9114.
[12] Scott O. Bradner. 1997. Key Words for Use in RFCs to Indicate Requirement Levels. Request for Comments RFC 2119. Internet Engineering Task Force. DOI: 10.17487/RFC2119.
[13] BuiltWith®. 2023. Root Authority Usage Distribution on the Entire Internet. https://trends.builtwith.com/ssl/root-authority/traffic/Entire-Internet.
[14] Andre Büttner, Hoai Viet Nguyen, Nils Gruschka, and Luigi Lo Iacono. 2021. Less is Often More: Header Whitelisting as Semantic Gap Mitigation in HTTP-Based Software Systems. In *ICT Systems Security and Privacy Protection*. DOI: 10.1007/978-3-030-78120-0_22.
[15] Stefano Calzavara, Sebastian Roth, Alvise Rabitti, Michael Backes, and Ben Stock. 2020. A Tale of Two Headers: A Formal Analysis of Inconsistent Click-Jacking Protection on the Web. In *USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity20/presentation/calzavara.
[16] Brian E. Carpenter. 1996. Architectural Principles of the Internet. Request for Comments RFC 1958. Internet Engineering Task Force. DOI: 10.17487/RFC1958.
[17] Jianjun Chen, Jian Jiang, Haixin Duan, Nicholas Weaver, Tao Wan, and Vern Paxson. 2016. Host of Troubles: Multiple Host Ambiguities in HTTP Implementations. In *ACM SIGSAC Conference on Computer and Communications Security*. DOI: 10.1145/2976749.2978394.
[18] MDN contributors. 2023. Evolution of HTTP. https://web.archive.org/web/20230821100712/https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP.
[19] [SW] Aldo Cortesi, Maximilian Hils, Thomas Kriechbaumer, and contributors, Mitmproxy: A Free and Open Source Interactive HTTPS Proxy version 7.0.2, 2021. URL: https://mitmproxy.org/.
[20] Dave Crocker and Paul Overell. 2008. Augmented BNF for Syntax Specifications: ABNF. Request for Comments RFC 5234. Internet Engineering Task Force. DOI: 10.17487/RFC5234.
[21] Martin Doyhenard. 2021. Response Smuggling: Pwning HTTP/1.1 Connections. HITB+ CyberWeek 2021. https://cyberweek.ae/2021/presentations/response-smuggling-pwning-http-1-1-connections/.
[22] Zakir Durumeric. 2023. Zakird/crux-top-lists: Downloadable snapshots of the Chrome Top Million Websites pulled from public CrUX data in BigQuery. https://github.com/zakird/crux-top-lists.
[23] [SW] Encode, HTTPX 2023. URL: https://www.python-httpx.org/.
[24] Let's Encrypt. 2023. Let's Encrypt Stats. https://letsencrypt.org/stats/.
[25] Adrienne Porter Felt, Richard Barnes, April King, Chris Palmer, Chris Bentzel, and Parisa Tabriz. 2017. Measuring HTTPS Adoption on the Web. In *USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/felt.
[26] Roy T. Fielding, Henrik Nielsen, Jeffrey Mogul, Jim Gettys, and Tim Berners-Lee. 1997. Hypertext Transfer Protocol – HTTP/1.1. Request for Comments RFC 2068. Internet Engineering Task Force. DOI: 10.17487/RFC2068.

[27] Roy T. Fielding, Mark Nottingham, and Julian Reschke. 2022. HTTP Semantics. Request for Comments RFC 9110. Internet Engineering Task Force. DOI: 10.17487/RFC9110.

[28] Roy T. Fielding, Mark Nottingham, and Julian Reschke. 2022. HTTP Semantics. Official Internet Protocol Standards STD 97. Internet Engineering Task Force. DOI: 10.17487/RFC9110.

[29] Roy T. Fielding, Mark Nottingham, and Julian Reschke. 2022. HTTP/1.1. Request for Comments RFC 9112. Internet Engineering Task Force. DOI: 10.17487/RFC9112.

[30] Roy T. Fielding and Julian Reschke. 2014. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. Request for Comments RFC 7230. Internet Engineering Task Force. DOI: 10.17487/RFC7230.

[31] [SW] Apache Software Foundation, Apache HTTP Server 2023. URL: https://github.com/apache/httpd.

[32] [SW] OpenJS Foundation, Node.Js 2023. URL: https://github.com/nodejs/node/tree/main.

[33] Gertjan Franken, Tom Van Goethem, Lieven Desmet, and Wouter Joosen. 2023. A Bug's Life: Analyzing the Lifecycle and Mitigation Process of Content Security Policy Bugs. In *USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity23/presentation/franken.

[34] Matthias Gierlings, Marcus Brinkmann, and Jörg Schwenk. 2023. Isolated and Exhausted: Attacking Operating Systems via Site Isolation in the Browser. In *USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity23/presentation/gierlings.

[35] Google. 2023. Chrome Platform Status. https://chromestatus.com/metrics/feature/popularity.

[36] Google. 2014. HTTPS as a ranking signal. Google for Developers. https://developers.google.com/search/blog/2014/08/https-as-ranking-signal.

[37] Mattias Grenfeldt, Asta Olofsson, Viktor Engström, and Robert Lagerström. 2021. Attacking Websites Using HTTP Request Smuggling: Empirical Testing of Servers and Proxies. In *IEEE International Enterprise Distributed Object Computing Conference*. DOI: 10.1109/edoc52215.2021.00028.

[38] Florian Hantke and Ben Stock. 2022. HTML violations and where to find them: a longitudinal analysis of specification violations in HTML. In *ACM Internet Measurement Conference*. DOI: 10.1145/3517745.3561437.

[39] Mario Heiderich, Jörg Schwenk, Tilman Frosch, Jonas Magazinius, and Edward Z. Yang. 2013. mXSS attacks: attacking well-secured web-applications by using innerHTML mutations. In *ACM SIGSAC Conference on Computer and Communications Security*. DOI: 10.1145/2508859.2516723.

[40] Charlie Hothersall-Thomas, Sergio Maffeis, and Chris Novakovic. 2015. BrowserAudit: automated testing of browser security features. In *International Symposium on Software Testing and Analysis*. DOI: 10.1145/2771783.2771789.

[41] Bahruz Jabiyev, Steven Sprecher, Anthony Gavazzi, Tommaso Innocenti, Kaan Onarlioglu, and Engin Kirda. 2022. FRAMESHIFTER: Security Implications of HTTP/2-to-HTTP/1 Conversion Anomalies. In *USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity22/presentation/jabiyev.

[42] Bahruz Jabiyev, Steven Sprecher, Kaan Onarlioglu, and Engin Kirda. 2021. T-Reqs: HTTP Request Smuggling with Differential Fuzzing. In *ACM SIGSAC Conference on Computer and Communications Security*. DOI: 10.1145/3460120.3485384.

[43] Jordan Jueckstock, Shaown Sarker, Peter Snyder, Aidan Beggs, Panagiotis Papadopoulos, Matteo Varvello, Benjamin Livshits, and Alexandros Kapravelos. 2021. Towards Realistic and Reproducible Web Crawl Measurements. In *The Web Conference*. DOI: 10.1145/3442381.3450050.

[44] James Kettle. 2019. HTTP Desync Attacks: Request Smuggling Reborn. PortSwigger Research. https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn.

[45] Sunwoo Kim, Young Min Kim, Jaewon Hur, Suhwan Song, Gwangmu Lee, and Byoungyoung Lee. 2022. {FuzzOrigin}: Detecting {UXSS} vulnerabilities in Browsers through Origin Fuzzing. In *USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity22/presentation/kim.

[46] Lukas Knittel, Christian Mainka, Marcus Niemietz, Dominik Trevor Noß, and Jörg Schwenk. 2021. XSinator.com: From a Formal Model to the Automatic Evaluation of Cross-Site Leaks in Web Browsers. In *ACM SIGSAC Conference on Computer and Communications Security*. DOI: 10.1145/3460120.3484739.

[47] Mike Kosek, Leo Blöcher, Jan Rüth, Torsten Zimmermann, and Oliver Hohlfeld. 2020. MUST, SHOULD, DON'T CARE: TCP Conformance in the Wild. In *Passive and Active Measurement*. DOI: 10.1007/978-3-030-44081-7_8.

[48] Balachander Krishnamurthy, Martin Arlitt, T Labs, Hewlett-Packard Laboratories, Park Avenue, and Florham Park. 2001. PRO-COW: Protocol Compliance on the Web-A Longitudinal Study. In *USITS*. https://www.usenix.org/conference/usits-01/pro-cow-protocol-compliance-web%E2%80%93-longitudinal-study.

[49] Meng Luo, Pierre Laperdrix, Nima Honarmand, and Nick Nikiforakis. 2019. Time Does Not Heal All Wounds: A Longitudinal Analysis of Security-Mechanism Support in Mobile Browsers. In *Network and Distributed System Security Symposium*. DOI: 10.14722/ndss.2019.23149.

[50] Seyed Ali Mirheidari, Matteo Golinelli, Kaan Onarlioglu, Engin Kirda, and Bruno Crispo. 2022. Web Cache Deception Escalates. In *USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity22/presentation/mirheidari.

[51] Hoai Viet Nguyen, Luigi Lo Iacono, and Hannes Federrath. 2019. Mind the cache: large-scale explorative study of web caching. In *ACM/SIGAPP Symposium on Applied Computing*. DOI: 10.1145/3297280.3297526.

[52] Hoai Viet Nguyen, Luigi Lo Iacono, and Hannes Federrath. 2019. Your Cache Has Fallen: Cache-Poisoned Denial-of-Service Attack. In *ACM SIGSAC Conference on Computer and Communications Security*. DOI: 10.1145/3319535.3354215.

[53] Hoai Viet Nguyen, Luigi Lo Iacono, and Hannes Federrath. 2018. Systematic Analysis of Web Browser Caches. In *International Conference on Web Studies*. DOI: 10.1145/3240431.3240443.

[54] Henrik Nielsen, Roy T. Fielding, and Tim Berners-Lee. 1996. Hypertext Transfer Protocol – HTTP/1.0. Request for Comments RFC 1945. Internet Engineering Task Force. DOI: 10.17487/RFC1945.

[55] Henrik Nielsen, Jeffrey Mogul, Larry M. Masinter, Roy T. Fielding, Jim Gettys, Paul J. Leach, and Tim Berners-Lee. 1999. Hypertext Transfer Protocol – HTTP/1.1. Request for Comments RFC 2616. Internet Engineering Task Force. DOI: 10.17487/RFC2616.

[56] Mark Nottingham. 2023. REDbot. https://redbot.org/.

[57] Jon Postel. 1980. DoD Standard Transmission Control Protocol. Request for Comments RFC 761. Internet Engineering Task Force. DOI: 10.17487/RFC0761.

[58] Jon Postel. 1981. Transmission Control Protocol. Request for Comments RFC 793. Internet Engineering Task Force. DOI: 10.17487/RFC0793.

[59] Jannis Rautenstrauch, Giancarlo Pellegrino, and Ben Stock. 2023. The Leaky Web: Automated Discovery of Cross-Site Information Leaks in Browsers and the Web. In IEEE S&P. DOI: 10.1109/sp46215.2023.10179311.

[60] [SW] Jannis Rautenstrauch and Ben Stock, HTTP Conformance Checker 2023. URL: https://github.com/cispa/http-conformance.

[61] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. 2020. Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies. In *Network and Distributed System Security Symposium*. DOI: 10.14722/ndss.2020.23046.

[62] Sebastian Roth, Stefano Calzavara, Moritz Wilhelm, Alvise Rabitti, and Ben Stock. 2022. The Security Lottery: Measuring Client-Side Web Security Inconsistencies. In *USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity22/presentation/roth.

[63] Vaspol Ruamviboonsuk. 2022. The 2022 Web Almanac: HTTP. 23. HTTP Archive. https://almanac.httparchive.org/en/2022/http.

[64] Len Sassaman, Meredith L. Patterson, and Sergey Bratus. 2012. A Patch for Postel's Robustness Principle. *IEEE Security & Privacy Magazine*, 2. DOI: 10.1109/msp.2012.31.

[65] Kaiwen Shen, Jianyu Lu, Yaru Yang, Jianjun Chen, Mingming Zhang, Haixin Duan, Jia Zhang, and Xiaofeng Zheng. 2022. HDiff: A Semi-automatic Framework for Discovering Semantic Gap Attack in HTTP Implementations. In *IEEE/IFIP International Conference on Dependable Systems and Networks*. DOI: 10.1109/dsn53405.2022.00014.

[66] Hendrik Siewert, Martin Kretschmer, Marcus Niemietz, and Juraj Somorovsky. 2022. On the Security of Parsing Security-Relevant HTTP Headers in Modern Browsers. In *IEEE Security and Privacy Workshops*. SecWeb. DOI: 10.1109/spw54247.2022.9833880.

[67] Snyk. 2022. URL confusion vulnerabilities in the wild: Exploring parser inconsistencies. Snyk. https://snyk.io/blog/url-confusion-vulnerabilities/.

[68] Marco Squarcina, Pedro Adão, Lorenzo Veronese, and Matteo Maffei. 2023. Cookie Crumbles: Breaking and Fixing Web Session Integrity. In *USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity23/presentation/squarcina.

[69] Web Tech Survey. 2023. Website technology checker. Web Technology Survey. https://webtechsurvey.com/.

[70] The Measurement Factory. 2018. Co-Advisor. https://web.archive.org/web/20211202131011/http://coad.measurement-factory.com/.

[71] The Measurement Factory. 2001. HTTP Compliance and W3C QA. https://www.w3.org/2001/01/qa-ws/pp/alex-rousskov-measfact.html.

[72] Martin Thomson and Cory Benfield. 2022. HTTP/2. Request for Comments RFC 9113. Internet Engineering Task Force. DOI: 10.17487/RFC9113.

[73] Martin Thomson and David Schinazi. 2023. Maintaining Robust Protocols. Request for Comments RFC 9413. Internet Engineering Task Force. DOI: 10.17487/RFC9413.

[74] W3C. 2023. Content Security Policy Level 3. Working Draft. https://www.w3.org/TR/2023/WD-CSP3-20230220/.

[75] W3C. 2022. Upgrade Insecure Requests. Editor's Draft. https://w3c.github.io/webappsec-upgrade-insecure-requests/.

[76] W3C. 2012. UriTesting. https://www.w3.org/wiki/UriTesting.

[77] W3Techs. 2023. Usage Statistics and Market Share of Web Servers, May 2023. https://w3techs.com/technologies/overview/web_server.

[78] Lukas Weichselbaum and Michele Spagnuolo. CSP-A Successful Mess Between Hardening and Mitigation. (2020). https://static.sched.com/hosted_files/

locomocosec2019/db/CSP%20-%20A%20Successful%20Mess%20Between%20Hardening%20and%20Mitigation%20(1).pdf.
[79] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. 2016. CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In *ACM SIGSAC Conference on Computer and Communications Security*. DOI: 10.1145/2976749.2978363.
[80] WHATWG. 2023. Fetch Standard. https://fetch.spec.whatwg.org/commit-snapshots/8f109835dcff90d19caed4b551a0da32d9d0f57e/.
[81] WHATWG. 2023. HTML Standard. https://html.spec.whatwg.org/commit-snapshots/578def68a9735a1e36610a6789245ddfc13d24e0/.
[82] WHATWG. 2023. MIME Sniffing Standard. https://mimesniff.spec.whatwg.org/.
[83] Seongil Wi, Trung Tin Nguyen, Jiwhan Kim, Ben Stock, and Sooel Son. 2023. DiffCSP: Finding Browser Bugs in Content Security Policy Enforcement through Differential Testing. In *Network and Distributed System Security Symposium*. https://www.ndss-symposium.org/wp-content/uploads/2023/02/ndss2023_f200_paper.pdf.
[84] WPT. 2023. Web-platform-tests documentation. https://web-platform-tests.org/.
[85] WPT. 2023. Wptserve: Web Platform Test Server — web-platform-tests documentation. https://web-platform-tests.org/tools/wptserve/docs/.
[86] Yuchen Wu and Andrew Hauck. 2022. How we built Pingora, the proxy that connects Cloudflare to the Internet. The Cloudflare Blog. http://blog.cloudflare.com/how-we-built-pingora-the-proxy-that-connects-cloudflare-to-the-internet/.
[87] Michal Zalewski. 2011. *The Tangled Web: A Guide to Securing Modern Web Applications*.

A SERVER DETAILS AND ADDITIONAL RESULTS

Table 5 shows all rules violated at least once in the wild and the number of unique hosts. The explanation and test for each rule is available online [60].
Table 6 shows the tested versions of each considered local server.

Type	Rule Name	#Hosts
Rec	STS after UIR*	8,440
Rec	Redirect after HTTP-UIR	7,822
Req	Allow header present for 405	7,406
Rec	HEAD and GET same headers	4,445
Rec	Accept-Patch if PATCH supported	3,419
Rec	Cookies use IMF-fixdate	2,850
Req	Content-Length for HEAD=GET	2,272
Req	Forbidden status-codes for POST	1,560
Req	Same headers for 304 and 200	1,145
Req	Date header required	663
ABNF	Experies ABNF	657
Req	STS not allowed for HTTP	550
Rec	Duplicate cookie names	518
Req	Duplicate headers	465
Rec	Content-Type header required	290
ABNF	XFO ABNF	262
Req	Mandatory headers for 206	217
ABNF	Etag ABNF	121
ABNF	Set-Cookie ABNF	93
ABNF	ACAO ABNF	83
Req	Duplicate STS	82
Rec	Duplicate CSP	71
ABNF	Accept-Patch ABNF	68
Req	WWW-Authenticate required for 401	63
Req	Forbidden Content-Length for 1XX and 204	55
ABNF	Last-Modified ABNF	52
Req	Forbidden surrounding whitespace for fields	51
ABNF	Content-Type ABNF	48
Req	Forbidden content for 304	46
ABNF	Server ABNF	34
ABNF	Date ABNF	30
ABNF	ACAC ABNF	28
ABNF	Vary ABNF	27
Req	Content-Length for 304=200	27
ABNF	CSP ABNF	23
Rec	Cookies with duplicate attributes	22
ABNF	Age ABNF	22
ABNF	Cache-Control ABNF	21
ABNF	Content-Language ABNF	20
ABNF	ACAM ABNF	19
ABNF	PermissionsPolicy ABNF	18
ABNF	ACAH ABNF	18
ABNF	STS ABNF	15
Rec	Content-Range required for 416	10
ABNF	XCTO ABNF	10
Rec	Location required for 302	9
Rec	Duplicate CSP-RO	7
Req	Proxy-Authenticate required for 407	6
Rec	Missing required headers for 415	6
ABNF	COOP ABNF	6
ABNF	Location ABNF	5
Rec	Overly long Server header	5
ABNF	CORP ABNF	3
ABNF	Content-Length ABNF	3
ABNF	ACMA ABNF	3
ABNF	Allow ABNF	3
Rec	Forbidden token form in no-cache directive	2
Rec	Location required for 301	2
Rec	Location required for 303	2
ABNF	Connection ABNF	2
Rec	Location required for 307	1
Req	Upgrade required for 101	1
Req	TE forbidden for non HTTP/1.1 responses	1
Req	TE forbidden for 1XX and 204	1
Req	Duplicate directives for STS	1
Req	Content-Range required for 206	1
ABNF	Range ABNF	1
ABNF	COEP ABNF	1

Table 5: Number of unique violating hosts per rule (* only applies to STS-safe hosts)

Server	Version
Apache	2.4.55
Caddy	2.6.4
Jetty	11.0.13
Nginx	1.21.4-1
Node	18.14.2
OpenLiteSpeed	1.7.16
OpenResty	1.21.4-1
Tomcat	10.1.5
Traefik	2.9.8

Table 6: Tested Server Versions.