Equivocal URLs: Understanding the Fragmented Space of URL Parser Implementations

Joshua Reynolds[1,2], Adam Bates[2], and Michael Bailey[2,3]

[1] New Mexico State University jr1@nmsu.edu
[2] University of Illinois at Urbana-Champaign {mdbailey,batesa}@illinois.edu
[3] Georgia Institute of Technology

Abstract. Uniform Resource Locators (URLs) are integral to the Web and have existed for nearly three decades. Yet URL parsing differs subtly among parser implementations, leading to ambiguity that can be abused by attackers. We measure agreement between widely-used URL parsers and find that each has made design decisions that deviate from parsing standards, creating a fractured implementation space where assumptions of uniform interpretation are unreliable. In some cases, deviations are severe enough that clients using different parsers will make requests to different hosts based on a single, "equivocal" URL. We systematize the thousands of differences we observed into seven pitfalls in URL parsing that application developers should beware of. Finally, we demonstrate that this ambiguity can be weaponized through misdirection attacks that evade the Google Safe Browsing and VirusTotal URL classifiers. URL parsing libraries have made a tradeoff to favor permissiveness over strict standards adherence in URL parsing. It is our hope this work will aid in motivating a systemic adoption of a more unified URL parsing standard enabling a more secure Web.

Keywords: URL · Parsing Ambiguity · Web Security

1 Introduction

Uniform Resource Locators (URLs) play a crucial role in the Internet, originating in the early 1990's as a standardized addressing and parameterization system for the Web [21, 39]. Since then, URLs have been overhauled to clarify their syntax with relation to relative locators [31], IPv6 addresses [32], Punycode for non-ascii hostnames [25], and the broader notion of a Uniform Resource Identifier (URI) [18, 19, 20] Further, the Web Hypertext Application Technology Working Group (WHATWG), a consortium of major Web browser vendors, has defined its own "living" URL parsing standard [4]. Unfortunately, adherence to these standards has not been strict, leading to inconsistencies across implementations when parsing some URLs.

Attackers have taken note of these inconsistencies and increasingly abuse URL parsing differences [55, 56, 9, 54, 58, 34, 41, 45]. In these exploits, attackers were able to trigger application-layer and network-layer vulnerabilities with

URLs parsing to a legitimate resource for one parser (e.g., a URL security classifier, a server endpoint) but a malicious resource for their victim (e.g., a browser, a server-side cache, etc.). To underscore the severity of this problem, consider Marlinspike's 2009 demonstration of canonical name parsing errors in TLS certificate generation infrastructure [44]; while this issue was resolved for TLS, parsing inconsistencies were exploited in a URL parser as recently as 2019 [58].

While anecdotal demonstrations of these "equivocal" URLs have appeared in industry reports, to date there has not been a systematic study of the root cause of this problem – inconsistent implementation of URL parsing. In this work, we measure the implementation space of URL parsing by analyzing the behavior of fifteen popular parsers. We focus on ambiguities in hostnames because of their potential impact at the network layer – sending clients with different URL parsers to completely different network locations. We generate and test thousands of fuzzing inputs to compare the level agreement of parsers with reference implementations, and among each other. Unfortunately, we find that disagreement is widespread, with little consensus on how to handle edge-case URLs. We then categorize the error sources that cause some URLs to only be parsable by certain parsers – or, worse, URLs that yield differing DNS-compliant hostnames for different parsers. We systematize these error sources into seven pitfalls that application developers need to beware of to avoid hostname equivocation.

To highlight the security implications of URL hostname equivocation, we go on to demonstrate how newly-discovered errors can allow equivocal URLs to evade URL classification. In contrast to prior work that has exclusively targeted server-side parsing errors at the application layer, we demonstrate that client-side URL security classifiers are also vulnerable. Specifically, *we demonstrate that URLs with ambiguous hostnames can trick the popular Google Safe Browsing and VirusTotal URL classifiers into issuing an incorrect threat classification.*

Fixing these inconsistencies among parsers would require community-wide agreement on a parsing standard whose strict implementation would be a breaking change. We perform preliminary measurements demonstrating the real-world compatibility incentive for URL parsers to eschew strict standardization in favor of being as permissive as possible in what they accept. We hope this work motivates the systemic adoption of a more unified URL parsing standard.

2 Related Work

URLs are composed of syntactic sections separated by delimiters. Figure 1 shows the syntactic segments that make up a URL. Schemes are defined by the Internet Assigned Numbers Authority (IANA) [3]. We focus in this work only on absolute URLs using the HTTP and HTTPS *Schemes*. Following the scheme is the optional *UserInfo* section, a *Hostname* or IP address, and an optional TCP *Port*. The *Path* commonly reflects a hierarchical naming system within a Web domain. *Queries* can carry parameters for which "&" and ";" are suggested as delimiters between parameters. Finally, *Fragments* are not sent in HTTP(S) requests, but are used by clients to locate specific portions of a resource after it is requested

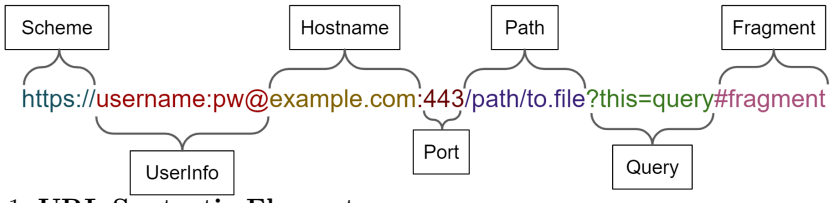

Fig. 1: **URL Syntactic Elements**
URLs use delimiters between each syntactic element. We find that URL parsers handle illegal characters and delimiters differently, yielding inconsistent results.

and received. Each URL segment has a limited character set, and octets outside that character set can be encoded with an escape sequence of a percent sign followed by two hexadecimal digits.

Driven by the need for interoperability with the Web, today there exists a broad ecosystem of URL parsing implementations. URL parsing libraries are standard issue with major programming languages. Further, various web clients, command line utilities, and web servers all implement their own URL parsers. The security of the Web depends, in part, on the basic assumption that all of these parsers will resolve a given URL in the same way.

2.1 Exploiting Human Misinterpretation of URLs

URLs can be made misleading to users, who fall prey to attacks like phishing. When users misunderstand the guarantees of HTTPS [10, 30, 28, 29, 52, 42], fail to observe the Fully Qualified Domain Name (FQDN) of a URL [57, 26, 13], or are unable to parse a URL [48, 11], attackers may convince them to reveal secrets by impersonating a legitimate organization. Phishing has been widely studied, and a host of mitigations have been designed to protect users from falling victim to these attacks. These include automatically phishing URL classifiers [15, 49, 60, 49, 40, 33, 12, 46, 14], phishing detection [51, 43], user education [37, 50, 36, 38, 22], and improved user interfaces [48, 11, 10, 30, 28, 29, 52, 43].

2.2 Exploiting Machines' Inconsistent URL Parsing

Unfortunately, phishing-like URL misinterpretations can also occur in software. URL parsing differences gained widespread attention in 2009 following Carretoni and di Paola's demonstration of HTTP Parameter Pollution attacks [24]. This attack abused differences in the parsing of URL query parameters between endpoints and security mechanisms, enabling attackers to bypass input filtering and sanitization checks. Subsequent prior works developed tools to automatically detect HTTP parameter pollution vulnerabilities in websites [17, 16, 23]. While query parameter parsing differences can have serious implications for application-layer security, they cannot affect the authenticity of the web server; in contrast, we demonstrate that hostname parsing differences enable equivocation about

web server identity. Further, as we will show in this work, lessons learned from parameter parsing attacks have not been applied to ambiguities URL hostname parsing.

More recently, several Blackhat talks and Common Vulnerabilities and Exposures (CVEs) have leveraged URL parsing ambiguity to perform *server-side* attacks. Tsai showed how inconsistent strategies for normalizing paths containing "../" allowed access to forbidden resources when combined with the ill-defined syntax for URL path parameters [56]. A bug in the Google Chrome browser on iOS in 2018 allowed websites to use the HTML 5 history API to change the origin of the tab and run in other Web origins [54]. Wang et al. showed they could misdirect OAuth redirections and evade allowlist filters using URL parsing discrepancies [58]. Kettle used the fact that browsers accept both backslashes and forward slashes as path delimiters to convince websites to poison their own HTML cache entries [34]. Ahmed reported a similar error in an Node package in CVE-2018-3774 [9]. Tsai and Leitschuh both used URL parsing ambiguities to trick server-side middleboxes to forward protocol-smuggled requests to resources they should not have been able to contact. [55, 41] Muñoz and Tsai reported parsing errors to curl which were patched [45, 55]. While these exploits provide anecdotal evidence of individual parsing problems, in this work we systematically explore the ecosystem of URL parsing ambiguities, testing many parsers to create a catalog of inconsistencies that point to a systemic issue in the ecosystem. Further differentiating us from prior work, we are the first to demonstrate that such attacks are possible on *client-side* URL classifiers, directly enabling enabling user attacks like phishing.

3 Methodology

To date, URL parsing exploits have been reported in the context of specific vulnerabilities and parsing implementations, but it is not clear to what extent inconsistencies in URL parsing are widespread. To gain a more comprehensive understanding of the ecosystem, we consider a diverse set of fifteen parsers that span standard libraries, web servers, and command line tools. Parsers were drawn from libraries written in popular languages (Java, Go, Ruby, JavaScript, Python, PHP, Perl, C/C++), tools (wget, curl), and web servers (Apache, NGINX); a complete description of these parsers can be found in Table 4 of Appendix A.

3.1 "Ground Truth" Reference Parsers

An important first step in our analysis is to establish a reasonable baseline for how parsers should behave. Naturally, one such baseline should be RFC 3986 [20] that defines the syntax for uniform resource identifiers (URIs), of which URLs are a subset. RFC 3986 provides a formal grammar, but not an implementation, so we used the grammar to create our own reference implementation for absolute URL parsing in Python3. We note that RFC 3986 rejects non-ASCII input and expects any disallowed bytes to be properly escaped before parsing.

In contrast, an equally valid baseline to consider is how major web software vendors handle URLs in their day-to-day experience, including URLs with non-ASCII characters. For such a baseline, we look to the WHATWG's "living" URL standard [4] informed by browsers day-to-day interaction with non-ACII URLs. WHATWG implicitly defines a standard for URL parsing by releasing a parsing algorithm along with a reference implementation of their parsing algorithm in JavaScript. We include this as our second reference parser.

3.2 Test Input Enumeration

For each of the fifteen parsers, we then applied a large set of URL test inputs and recorded each parser's response. We focus specifically on parsing discrepancies in the hostname field. To do so, we started with a completely valid URL containing a hostname that was consistently parsed across all implementations. We then applied three mutually exclusive sets of mutations to this URL to enumerate a large corpus of test inputs. Each mutation inserts one to four bytes in the middle of the hostname field, as described below. Rather than using random fuzzing, we iterated over these sets in their entirety, resulting in a total of 98,425 test cases.

P1. The first input set inserted every possible octet from 0-255, which includes all standard ASCII codes (0-127) and extended ASCII codes (128-255). This test set probes parsers' permissiveness of invalid input as well as handling of duplicate delimiters.

P2. The second input inserted all 65,536 possible combinations of two octets. This test set further tests delimiter confusion by probing with an additional random byte, as well as the handling of valid and partial unicode characters.

P3. The third input set inserted each of the 32,634 valid Unicode code points listed in the Unicode Data list of the Unicode Character Database [7]. Each valid unicode character is a minimum of three bytes when encoded with UTF-8. This test set exhaustively probes parsers' handling of unicode characters.

Along with different parsing logic, our test parsers were also written in a variety of languages and software environments with different implementations of character strings, file I/O, etc. For each parser's testing apparatus, we took great care to ensure that the core parsing logic handled the exact same bytes for each test input. Mostly, this entailed paying close attention to how different string data types might apply automatic character conversions, although we note that in practice these differences are another potential source of parsing ambiguity. For two parsers we were forced to cast our payload URLs into string types that cannot hold arbitrary bytes to be compatible with the library. We have noted this conversion in our full parser list in Table 4 of Appendix A.

Parser	**Overall**	**P1** (256)	**P2** (65,536)	**P3** (32,635)
rfc3986	100.0%	100.0%	100.0%	100.0%
Ruby uri	99.95%	100.0%	99.93%	100.0%
PHP parse_url	97.79%	94.14%	96.7%	100.0%
Python3 urllib.urlparse	90.67%	82.81%	86.06%	100.0%
WHATWG NodeJS	59.74%	83.2%	83.73%	11.38%
Python3 furl	51.62%	51.95%	75.89%	2.88%
Golang goware/urlx	30.08%	44.92%	45.01%	0.0%
Java.net.URI	28.74%	49.61%	42.94%	0.05%
Golang net/url	26.61%	47.66%	39.78%	0.0%
libcurl4-openssl	23.19%	44.92%	34.66%	0.0%
wget	22.61%	44.53%	33.78%	0.0%
nginx	7.44%	32.03%	11.05%	0.0%
Apache Portable Runtime	7.2%	32.03%	10.69%	0.0%
Perl URI	6.68%	31.64%	9.92%	0.0%
NodeJS Legacy	4.94%	26.95%	7.32%	0.0%

Table 1: **Agreement with RFC 3986 Parser**
We show agreement with our RFC reference parser across URLs perturbed by input sets P1, P2,and P3 as well as overall. Parsers are sorted by their overall agreement with the standards of RFC 3986. Ruby and PHP follow the RFC with a high degree of consistency, but the remainder of the parsers are clearly not matching RFC 3986's grammar.

4 Results

We now report on the results of our analysis in terms of *agreement* between parsers on each of the test inputs. We consider two axes of agreement – consistency with the reference parsers, and overall consistency across all fifteen parsers – for the *UserInfo*, *Host*, *Path*, *Query*, and *Fragment* segments of each test input. To ensure a conservative analysis, we adopt a generous definition of what it means for two parsers to agree on a URL's parse. If both parsers rejected the URL with an error, we consider this as agreement regardless of whether the same error is thrown. We also ignore whether the parser includes the delimiter of a syntactic segment (e.g, '`/`' in '`'/index.html'`'). Because DNS is not case-sensitive, we also ignore hostname case in the parser output.

4.1 Disagreement with Reference Parsers

After testing each parser on all test inputs, we then sorted them by their level of agreement with each of the reference parsers. Agreement with RFC 3986 is given in Table 1. Only three of the parsers are often in agreement with the RFC – Ruby URI (99.9%), PHP's *filter_var()* plus *parse_url()* functions (97.79%), and Python3's urllib.urlparse (90.67%). However, even among these high-agreement parsers, we observed differences on how strict or lenient they were on certain

Parser	**Overall**	**P1** (256)	**P2** (65,536)	**P3** (32,635)
WHATWG NodeJS	100.0%	100.0%	100.0%	100.0%
Python3 urllib.urlparse	66.95%	97.27%	94.51%	11.38%
PHP parse_url	58.09%	78.91%	81.27%	11.38%
Ruby uri	59.78%	83.2%	83.8%	11.38%
rfc3986	59.74%	83.2%	83.73%	11.38%
Python3 furl	49.77%	51.56%	73.11%	2.88%
NodeJS Legacy	36.1%	27.73%	9.98%	88.62%
Golang goware/urlx	21.67%	28.91%	32.11%	0.64%
Java.net.URI	19.9%	32.81%	29.42%	0.69%
Golang net/url	19.19%	33.98%	28.36%	0.64%
libcurl4-openssl	17.92%	35.55%	26.46%	0.64%
wget	17.57%	35.16%	25.93%	0.64%
nginx	15.19%	46.09%	22.32%	0.64%
Apache Portable Runtime	15.17%	46.48%	22.28%	0.64%
Perl URI	14.58%	46.09%	21.72%	0.0%

Table 2: **Agreement with WHATWG Parser**
We show agreement with the WHATWG reference parser across URLs perturbed by input sets P1, P2,and P3 as well as overall. Not only are parsers not following RFC 3986, they are also not following the WHATWG's alternative parsing algorithm that handles international characters in URLs. The WHATWG parsing algorithm disagrees with most of these tested parsers most of the time.

inputs. For example, Ruby URI allows some octets outside the permitted character set such as low ASCII bytes in the query string. PHP was sometimes more strict and rejects some hostnames that could not be used with the DNS but are allowed by RFC 3986 containing characters like tilde and asterisk. However, PHP does allow a fragment to contain another illegal # delimiter. Python3 was more lenient, parsing URLs with illegal low ASCII bytes like 0x10 (newline) in the hostname. The WHATWG reference parser agreed with RFC 3986 just 59.74% of the time, while the remaining ten parsers agreement ranges from 51.62% (Python3 `furl`) to as low as 4.94% (NodeJS Legacy).

Agreement with the WHATWG parser is given in Table 2. Overall, the test parsers agreed more often with the older RFC than with the WHATWG's living standard. In fact, the RFC 3986 parser and the three high-agreement parsers from the previous test again boast the highest agreement with WHATWG.

Parsers are not following the WHATWG standard when handling Unicode. Recall that the P3 set tests parsers handling of UTF-8 encoded Unicode characters. Interestingly, even though one of the goals of the WHATWG parser is to standardize handling of non-ASCII URLs, agreement on the P3 input set is very poor. In fact, with the exception of NodeJS Legacy (88.62%), most parsing agreement on P3 only negligibly improves with WHATWG over RFC 3986, which does not support Unicode at all.

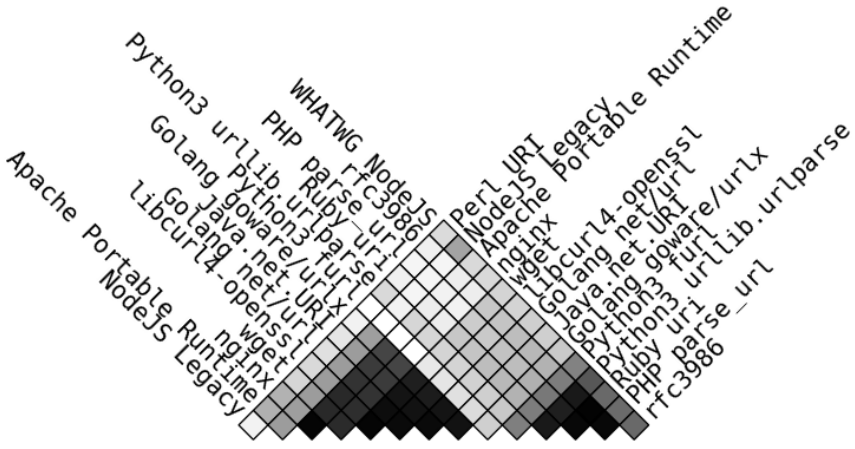

Fig. 2: **Pairwise Agreement Among Tested URL Parsers**
Intersections of parsers are shaded according to their overall agreement on a linear gradient from white (0% agreement) to black (100% agreement). No agreement was perfect, but we observe two dark clusters of high correlation at the bottom of the figure. One is centered around RFC 3986 and the other included Java, Golang, libcurl4, NGINX, Apache, and wget. The Furl, Perl, NodeJS Legacy, and WHATWG parsers do not closely match any other tested parser.

4.2 Disagreement Among All Parsers

Since the parsers do not consistently follow either reference implementation, it may be the case that parsing behavior is dictated by some other hidden standard or common practice. This would explain the Table 2 results in which many parsers disagree with the WHATWG parser at a very similar rate for the P3 test set (0.64% agreement). To investigate, we next calculated the pairwise agreement between each of the fifteen parsers for the combined test set of 98,445 URLs. This resulted in a total of 105 agreement calculations for each of the unique parser pairings.

We illustrate the agreement among all these pairs in Figure 2, with *darker* shading indicating *more* agreement and *lighter* shading indicating *less* agreement. The two dark triangles towards the bottom of the figure indicate two families of similarly behaving parsers. To the right-hand side, it can be seen that Ruby, Python3 urllib.urlparse, and PHP strictly adhere the RFC 3986 reference parser. The cluster to the left-hand side consists of Java.net.URI, the Golang parsers, NGINX, Apache, wget, and libcurl4. Like the first cluster, their agreement is not perfect, but is highly similar. Upon investigation, this cluster behaves differently from RFC 3986 because it accepts invalid octets into the hostname without throwing an error; this approach allows for internal consistency because such octets are not delimiters, and the illegal bytes could be converted to Punycode

later. In contrast, RFC 3986, or even the more permissive RFC 3987, would automatically escape illegal bytes or reject URLs with invalid octets. Finally, unlike WHATWG, this cluster does not accept illegal octets lower than the acceptable ASCII character range. Overall, this experiment underscores the fractured nature of the space of URL implementations, where arbitrary and invalid inputs are handled very differently depending on which parser is chosen.

5 A Taxonomy of URL Parsing Pitfalls

Having shown the extent of disagreement between URL parsers, we now consider the root causes of this disagreement. We first grouped each URL test input by the sets of parsers that agreed on its hostname; for example, two URLs would form a group if they caused errors in the same six parsers and were parsed to the same hostname by the remaining nine. From our 98,445 URLs, we created 134 groups using this method. Because many groups described inputs that results exclusively in errors and DNS-incompatible hostnames, we further down-selected to 17 groups for which there were at least two DNS-compatible hostnames in the results set. We then manually inspected each group to understand the cause of the inconsistency.

Ultimately, we arrived at a taxonomy of just seven potential URL parsing pitfalls that account for the all of the hostname equivocation inconsistencies observed in our experiments. We describe each pitfall in the remainder of this section, providing examples of each in Table 3. We also report on the effects of these equivocal URLs on the Chrome and Firefox browsers, as well as their embedded JavaScript engines. Browser design choices in this space will prove important in our examples of malicious equivocal URLs in Section 6.

5.1 Seven Pitfalls of URL Parsing Causing Hostname Equivocation

Pitfall 1: Null Bytes

In C, strings are traditionally terminated by a null byte. In higher-abstraction languages, the length property is often explicitly tracked, allowing strings containing null bytes. This technicality enabled Marlinspike's 2009 equivocations of subject names in TLS certificates [44].

URL parsers behave differently due to this same split when faced with a URL containing an illegal null byte. URL 1 in Table 3 is an equivocal URL built on this discrepancy. It places a null byte in the UserInfo section of the URL. The Golang, Java, PHP, and Ruby parsers correctly reject URL 1 as a malformed URL. On the other hand, Perl, PHP, and Python3 are willing to parse this URL and consider the null byte as part of the UserInfo. This set of parsers therefore considers this URL to point to "t.co". C-language based parsers including libcurl4, wget, and the Apache Portable Runtime (APR) truncate URLs to the first null byte because of their built-in null-terminated string assumptions. However, because NGINX uses a custom string implementation to track string length, it is not subject to this pitfall despite being written in C.

I. Equivocal URL Examples

Equivocal URL Example	Option A	Option B
U1. https://n.pr[0x00]@e.gg	e.gg	n.pr
U2. https://n.pr\@e.gg	e.gg	n.pr
U3. https://n.pr][e.gg	e.gg	n.pr
U4. https://n.pr#@e.gg	n.pr	e.gg
U5. https://n.pr%2ee.gg	n.pr.e.gg	n.pr
U6. https://n.pr[0x0A]e.gg	n.pre.gg	n.pr
U7. https://n.pr[0xDD9ADCBD]e.gg	n.xn–pre-hwf8l.gg	n.xn–pre-bda9o3gf.gg
U8. https://n.pr[0xC4B0]@e.gg	n.xn–prie-swc.gg	n.xn–pre-tfa3h.gg
	C: n.prie.gg	**D:** n.xn–pre-tfa3x.gg

II. Results of Parsing Each Equivocal URL

Parser	**U1**	**U2**	**U3**	**U4**	**U5**	**U6**	**U7**	**U8**
NodeJS WHATWG	A	B	ERR	A	A	A	ERR	A
RFC 3986	ERR	ERR	ERR	ERR	err	ERR	ERR	ERR
Golang net/url	ERR	ERR	err	A	ERR	ERR	err	err
Golang goware/urlx	ERR	ERR	ERR	A	ERR	ERR	err	C
Java.Net.URI	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
PHP parse_url	ERR	ERR	ERR	A	ERR	ERR	ERR	ERR
Python3 urllib	A	A	A	A	ERR	err	ERR	ERR
Python3 furl	A	A	err	A	ERR	err	err	err
NodeJS legacy	A	B	B	A	B	B	A	A
Ruby	ERR	ERR	ERR	ERR	err	ERR	ERR	ERR
wget url.c	B	A	err	A	A	ERR	err	err
libcurl4	B	A	err	A	err	ERR	err	err
Perl URI	A	A	err	A	A	err	B	B
Firefox	-	B	ERR	A	A	A	ERR	A
JS in Firefox	A	B	ERR	A	A	A	ERR	D
Chrome	-	B	ERR	A	A	A	ERR	A
JS in Chrome	A	B	ERR	A	A	A	ERR	D
Apache	B	A	err	A	err	err	err	err
NGINX	ERR	err	err	B	err	ERR	ERR	ERR

Table 3: **Equivocal URL Examples with Parsing Results**
In part **I**, eight examples of equivocal URLs are provided that, when parsed, yield at least two different DNS-compatible hostnames. Square brackets in these examples enclose a hex representation of octet(s) for clarity. In part **II**, parsing results for each example are provided. A, B, C and D indicate which option from part **I** each parser returned. Capital "ERR" signifies that the indicated parser threw an error for that example. Lowercase "err" indicates the parser did not throw an error, but extracted a hostname that is not compatible with the DNS. The only parser to throw an error for every example here is Java's URI parser.

Pitfall 2: Backslash Correction

URL 2 in Table 3 uses another illegal byte in a UserInfo section. Browsers have a particular treatment for the illegal \ character, making themselves ambivalent to the difference between Windows and *nix file path separators. When a backslash is present in a UserInfo section, it is treated as if it were a delimiter signalling the start of the URL path. This means that a backslash will be corrected to a forward slash when pasted into a browser. Illegal backslashes cause either an error or are treated as a part of the UserInfo in all parsers except the browsers. Both Firefox and Chrome change the backslash to a delimiting forward slash, which makes what was formerly a UserInfo string into the hostname. RFC 3986 considers this an invalid URL because a backslash is not allowed anywhere. The WHATWG parser is more permissive, and accepts the UserInfo section as-is. Several other parsers either allow or automatically encode the backslash and accept the URL. This pitfall has so far been exploited several times [58, 41, 9].

Pitfall 3: Overeager Percent Decoding

The official way to include arbitrary bytes in URLs is to URL-encode them in triplets of the form %FF with a percent sign followed by two case-insensitive hex digits. These digits are allowed in hostnames, but will not be compatible with the DNS as such. The example in URL 5 of Table 3 encodes one of the otherwise allowed periods in the hostname with percent encoding. When parsed, browsers and Perl convert this automatically into a period. The legacy JavaScript parser simply terminates the hostname at the percent-encoded section.

In context of HTTP requests, there are further complications regarding percent decoding that we will explore later on. We exploit the fact that JSON also uses percent-encoding to encode arbitrary bytes in strings to introduce ambiguities into API calls in Section 6.

Pitfall 4: IPv6+ Address Syntax

Square brackets are only allowed in hostnames to enclose IPv6 addresses or future IP versions. By inserting a balanced, but not matching set of brackets into the hostname, we convince the python3 parser and the legacy NodeJS parser to provide two different, DNS-compatible hostnames. For the example we gave in URL 3 of Table 3, Python3 starts the hostname after the brackets, and NodeJS's legacy parser truncates the hostname at the brackets.

Pitfall 5: Automatic Punycode Conversion

While some parsers simply reject invalid URLs, others try to fix them. Some of these parsers use Punycode [25] to encode arbitrary bytes in hostnames within the syntactic bounds rules of the DNS. We found that some parsers disagreed

on how they would perform this conversion. Example URLs 7 and 8 in Table 3 were encoded in several different ways, depending on the parser. Example 8 is unique among our examples in that it resolves to four different DNS-compatible domain names depending on the parser. These problems manifested for Unicode issues such as how to convert an arbitrary character to lowercase, or how to deal with only half of a surrogate pair. In general, users of URL parsers would do well to consult the Unicode Security Guide [59]. This pitfall was among those exploited by Tsai [55].

Pitfall 6: Low ASCII Bytes

Another set of illegal octets that parsers treat differently in a UserInfo section is the set of octets lower than any allowed character. As demonstrated by URL 6 in Table 3, allowing the ASCII newline, 0x0A, yields different interpretations. Golang, Java, and Ruby's parsers correctly reject URL 6 as a malformed URL. All other parsers accept this URL, and either ignore or treat the newline byte as part of the UserInfo with the true host being "e.gg". However, in Perl, the newline is ignored and the UserInfo is pre-pended to the hostname.

Pitfall 7: Illegal Extra Delimiters

Some parsers allow ambiguity by allowing prohibited extra delimiters such as allowing an "@" in a UserInfo or multiple "#" characters. URL 4 in Table 3 is an example of a URL with a duplicate delimiter. Many of the parsers we tested allowed this incorrect choice. Fragments in particular were very permissive. The pitfall was also part of Tsai's exploit [55].

6 Misdirection Attacks with Equivocal URLs

While we have demonstrated widespread inconsistencies in URL parsing behavior, we have not yet demonstrated whether these equivocal URLs represent a pressing security concern. We now show how equivocal URLs can be weaponized by an attacker who can anticipate the parsing libraries in use on a victim's system. Specifically, we demonstrate how equivocal URLs can cause false negatives in the Google Safe Browsing and VirusTotal URL classifier services [6, 8] through the creation of URLs that parse to a legitimate host in the security software but a malicious host in the victim software.

Threat Model: In this work, we consider an adversary whose goal is to cause a victim program to fetch a malicious resource by making it appear to a URL classifier as if the URL came from a trusted domain. The adversary can take advantage of differences in URL parsing behavior between the victim program and the URL classifier protecting it. In this example, the victim uses classifiers provided by VirusTotal (VT) [8] and Google Safe Browsing (GSB) [6] to evaluate

URLs before requesting a resource using a URL. We consider both the Web interfaces and API endpoints of these classifiers, which do not behave identically.

6.1 Responsible Disclosure

We informed both services of these ambiguities in October of 2021. Unfortunately, we have not received any response beyond a request to forward our report as a feedback ticket. We promptly complied with the request, but these ambiguities persist in these systems.

6.2 Equivocal URLs vs Google Safe Browsing

To demonstrate equivocal URLs' ability to cause a false negative, we first need a known-malicious URL from GSB to prove equivocal URLs can have this real-world effect. Fortunately, GSB has a test vector URL which is always flagged as malware "`https://malware.testing.google.test/testing/malware/*`". By leveraging Pitfall 3 (over-eager percent decoding), we are first able to craft an equivocal URL that convinces the GSB API to classifier a URL as clean even though it parses to the test vector when loaded in a browser. Consider the following equivocal URL, remembering that %2F encodes an ASCII forward slash:

```
http://letsencrypt.org%2F@malware.testing.google.test/testing/malware/*
```

GSB's API passes URLs in JSON. JSON's specification allows percent-encoding bytes in strings. Therefore, when this URL arrives at GSB, GSB has no way of knowing whether the %2F is intended as a literal delimiter, or a percent-encoded portion of the UserInfo. GSB chooses the former, and reports this URL is clean in both its web interface and API. However, this syntactically valid URL will lead the other clients we tested to target malware.testing.google.test. We also discovered that this API performs the same "backslash correction" as browsers. Interestingly, a percent-encoded backslash (%5C) in the UserInfo will be decoded and then trigger backslash correction in the API. As an example of this backslash correction, the following URL is also declared safe by GSB's API:

```
http://letsencrypt.org%5C@malware.testing.google.test/testing/malware/*
```

However, the Web interface does not have this eager percent decoding functionality nor backslash correction. But, knowing this behavior we can craft an equivocal URL which the Web interface declares safe, but would send a browser to the malware test vector. Because the Web interface fails to account for the pitfall we called "backslash correction" in browsers, it evaluates the benign host while a browser would fetch the malware:

```
https://malware.testing.google.test\testing\malware\*@letsencrypt.org
```

The fractured landscape of parsers creates a dilemma for security systems like GSB. No matter how GSB parses an equivocal URL, there exist other parsers that would extract a different hostname. We present some potential mitigation strategies in Section 8.1.

6.3 Misdirecting VirusTotal

Using a similar approach, we are also able to create an equivocal URL that fools VirusTotal's URL scanning web endpoint and API. For the benign domain, we again used "letsencrypt.org". For a malicious domain, we referenced urlHaus's public list of online URLs serving malware [5]. The URL we selected served malware flagged by fifteen of VirusTotal's 90 constituent scanners.

Inputs to VirusTotal's API are form-encoded in a post request, and thus similar ambiguity exists for their endpoint as to whether or not percent-encoded delimiters should be reconstituted. We take advantage of this to create a URL that uses "overeager percent decoding" to cause VirusTotal to report our malicious URL as totally clean. The following URL is a false negative for both the Web interface and the API. The actual malicious host used has been redacted.

```
http://letsencrypt.org%2Fdocs%2F@[redacted]/LS.exe
```

In testing other equivocal URL techniques, we observed that we were able to pacify some of the original fifteen alerting constituent classifiers. This suggests that they each are vulnerable to equivocal URLs in their own way – depending on the parsing or matching strategy they use to compare URLs. Adding any UserInfo string pacified two scanners, suggesting that this may have evaded an internal blocklist. A third classifier was pacified if that UserInfo contained a null byte. Three others appear to perform backslash correction. The inconsistency among constituent classifiers when faced with a URL change is cause for concern.

7 Backwards Compatibility Constraints on Strict URL Parsing

Blindly mandating strict parser adherence to a new or existing URL parsing standard would likely break some services. While a full measurement of what services would be impacted by stricter URL parsing at Web scale would require its own paper, we can give some preliminary estimations here for the upper bound of the impact. We do this by repurposing several public data sources to learn how often services' URLs use non-ASCII characters. Ecosystem-wide standardization of URL parsing would affect some fraction of these services, making them an approximate upper bound on the potential impact.

We first find the prevalence of Punycode enabling non-ASCII characters in domain names. Among the Alexa Top Million [1] list, 0.16% (1,606) of domains use Punycode [25] to encode non-ASCII hostnames. Of the 7.8 billion TLS certificates available in Censys's database [27], 0.41% (32,342,256) use Punycode in their subject or alternative domain names.

We also surveyed ∼350 million URLs sampled uniformly and randomly from the approximately 3 billion URLs in Common Crawl's January 2022 URL Index [35]. Of these URLs, 0.04% contained unicode characters that were left to the client to parse when making an HTTP(S) request. By contrast, 9.95% of these URLs had escaped their problematic bytes themselves with percent-encoding.

These Web applications, it seems, have elected to perform percent encoding for themselves. Perhaps these applications have an understanding of the compatibility risks of relying on an arbitrary client's URL parser.

While the overall percentage of services that would be affected by unified parsing standardization appears to be low, this still implicates a large number of services at Web scale. At the same time, it preliminarily appears that a path forward to consistent URL parsing standard is possible with minimal impact on existing services.

8 Discussion

In the space of URL parsing, we have become too liberal in what we are willing to accept. Jon Postel's "Robustness Principle" [47] promotes compatibility at the expense of correctness. In this case, the cost of compatibility also gives rise to concerns of security and authenticity on the Web. In fact, as noted in an 2018 IETF draft [53], Jon Postel wrote his famous remark on conservative sending and liberal receiving immediately following this sentence:

> "*While the goal of this specification is to be explicit about the protocol there is the possibility of differing interpretations [47].*"

Perhaps to call these departures from URLs' specification "differing interpretations" is too generous, but the fact remains that the today's ecosystem of URL parsers is in broad disagreement with itself. Standardizing the myriad URL parsing libraries, which are baked into nearly every piece of network software, would be a massive undertaking requiring the cooperation of many stakeholders. Such uniformity might not be backwards compatible. Certainly, efforts like the WHATWG's URL living standard are evidence of a desire to eventually correct URL ambiguity. However, even this formidable consortium of leading Web browser creators has not brought uniformity.

We note that some parsers even document the risks of yielding different results than other parsers. The documentation of PHP's *parse_url* function includes the following warning:

> "**Caution** *This function may not give correct results for relative or invalid URLs, and the results may not even match common behavior of HTTP clients.*"[2]

The documentation then proceeds to explain a method to enforce stricter parsing, which we made use of, meaning that the default behavior is even more permissive than what we report. A developer who did not read the documentation would be unaware of these edge-cases.

8.1 Mitigation

Individual parsers are limited in their ability to correct these systemic inconsistencies alone. Some already include optional flags to perform stricter parsing.

However, they should collectively choose to create or follow a common parsing algorithm such as the regularly updated, "living" WHATWG standard [4]. Such changes would technically be breaking compatibility for downstream Web software. However, our initial investigation in Section 7 suggests that the services depending on the particulars of these edge-case parsing implementations may be uncommon.

For the present, we recommend that security tools analyzing URLs take note of the potential ambiguity that exists between their filter's URL parsers and the URL parsers of clients they protect. Ultimately, we hope URL parser implementations will unite around an updated URL standardization that is acceptable to all. In the meantime, one possible avenue of protection would be to apply a strict interpretation of RFC 3986's grammar [20], and fail closed. Another would be to align URL classifiers with the classifiers they intend most often to protect. For example, Google Safe Browsing's main purpose is to flag Web sites unsafe to visit in a browser, and could benefit from aligning its parser with the WHATWG's parser. A third option would be to create a multi-parser to simultaneously parse equivocal URLs in various ways, allowing a URL classifier to check each potential interpretation.

8.2 Limitations and Future Work

This work does not test all extant URL parsers, nor does it exhaustively exercise every code path within each parser to find every inconsistency. Rather, we demonstrate the existence of parsing ambiguity across a variety of parsers and demonstrate how that ambiguity can be used to obscure a URL's destination. Given the level of inconsistency we observed, we are confident that further testing would only magnify the extent of disagreements between URL parsers.

Future work may expand this set of URL parsing pitfalls and their effects on more systems. Likewise the parsing and filtering behaviors of popular antivirus and middleboxes should be checked for blind spots where equivocal URLs are concerned. Deeper measurement should be done to design a consistent standard that all parsers could adopt with minimal impact to existing services.

9 Conclusion

Given the fragmented implementation space of URL parsers, we warn that parsing inconsistencies among mainstream URL parsers continues to be an active attack vector. Because of the ubiquity of Web connectivity in modern applications, developers would do well to be aware of and plan for URL parser discrepancies until uniformity is achieved.

Acknowledgements This work was partially supported by NSF under grant GR0005987. We thank Zane Ma, Joshua Mason, Kent Seamons, Jay Misra, Kaylia M. Reynolds, Deepak Kumar, and Paul Murley for their feedback and suggestions.

Appendix A: Tested Parser Details

Parser Name	Version	Language	Category	Parser Input Type	Type Coercion Applied
RFC 3986	-	Python 3.8.10	Control	bytes	none
WHATWG Reference Parser	-	NodeJS 10.19.0	Control	JS Buffer	none
Python3 urllib	-	Python 3.8.10	Built-In Libraries	bytes	none
parse_url with filter_var()	-	PHP 7.4.3	Built-In Libraries	PHP string	none
NodeJS Legacy	-	NodeJS 10.19.0	Built-In Libraries	JS String	none
Java.Net.URI	-	openjdk 17.0.1	Built-In Libraries	Java String	UTF-8 decoding
Ruby uri	0.10.0	ruby 2.7.0p0	Built-In Libraries	Ruby String	none
Golang net/url	-	Golang 1.13.8	Built-In Libraries	golang-string	none
libcurl4	7.68.0	C	Unix Tools	char*	none
wget	1.21	C	Unix Tools	char*	none
perl URI	-	Perl 5.30.0	Unix Tools	perl-string	none
Apache Portable Runtime	httpd-2.4.48	C/C++	Open Source Parsers	char*	none
NGINX	1.20.0	C	Open Source Parsers	char* & length	none
fURL	2.1.3	Python 3.8.10	Open Source Parsers	Python3 string	UTF-8 decoding
Golang goware/urlx	dcd04f6	Golang 1.13.8	Open Source Parsers	golang-string	none

Table 4: **Parsers Tested**
We tested our URLs in these 15 URL parsers. While most parsing libraries accepted input types containing arbitrary bytes, Java's URL parser and the fURL parser both required the sequence of bytes to be converted to a string type. We followed Python3's default behavior to throw an error upon encountering bytes it cannot decode with UTF-8, and Java's default behavior to replace bytes not valid in UTF-8 with marker character 0xEFBFBD. Where larger systems like NGINX are listed, we extracted URL parsing functionality from source code.

References

1. Alexa top 1,000,000 sites. `http://s3.amazonaws.com/alexa-static/top-1m.csv.zip`
2. parse_url. php.net (2021)
3. Uniform resource identifier (uri) schemes. IANA (2021)
4. Url: Living standard. Web Hypertext Application Technology Working Group (2021)
5. Urlhaus. abuse.ch (2021)
6. Google safe browsing. `safebrowsing.google.com` (2022)
7. Unicode character database: Unicodedata.txt. `https://www.unicode.org/Public/UCD/latest/ucd/UnicodeData.txt` (2022)
8. Virus total. `www.virustotal.com` (2022)
9. Ahmed: url-parse package return wrong hostname. Hackerone (2018)
10. Akhawe, D., Felt, A.P.: Alice in warningland: A large-scale field study of browser security warning effectiveness. In: 22nd {USENIX} Security Symposium ({USENIX} Security 13) (2013)
11. Albakry, S., Vaniea, K., Wolters, M.K.: What is this url's destination? empirical evaluation of users' url reading. Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (2020)
12. Aljofey, A., Jiang, Q., Qu, Q., Huang, M., Niyigena, J.P.: An effective phishing detection model based on character level convolutional neural network from url. Electronics (2020)
13. Alsharnouby, M., Alaca, F., Chiasson, S.: Why phishing still works: User strategies for combating phishing attacks. International Journal of Human-Computer Studies
14. Althobaiti, K., Rummani, G., Vaniea, K.: A review of human-and computer-facing url phishing features. In: 2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). IEEE (2019)
15. Anitha, A., Gudivada, K.S., Rakshitha Lakshmi, M., Kumari, S., Usha, C.: Identifying phishing websites through url parsing
16. Athanasopoulos, E., Kemerlis, V.P., Polychronakis, M., Markatos, E.P.: Arc: protecting against http parameter pollution attacks using application request caches. In: International Conference on Applied Cryptography and Network Security (2012)
17. Balduzzi, M., Gimenez, C.T., Balzarotti, D., Kirda, E.: Automated discovery of parameter pollution vulnerabilities in web applications. In: NDSS (2011)
18. Berners-Lee, T.: Rfc 1630: Universal resource identifiers in www: a unifying syntax for the expression of names and addresses of objects on the network as used in the world-wide web (1994)
19. Berners-Lee, T., Fielding, R., Masinter, L.: Rfc 2396: Uniform resource identifiers (uri): generic syntax (1998)
20. Berners-Lee, T., Fielding, R., Masinter, L.: Rfc 3986: Uniform resource identifier (uri): Generic syntax (2005)
21. Berners-Lee, T., Masinter, L., McCahill, M.: Rfc 1738: Uniform resource locators (url) (1994)
22. Canova, G., Volkamer, M., Bergmann, C., Reinheimer, B.: Nophish app evaluation: lab and retention study. In: NDSS workshop on usable security (2015)
23. Cao, Y., Wei, Q., Wang, Q.: Parameter pollution vulnerabilities detection study based on tree edit distance. In: International Conference on Information and Communications Security. Springer

24. Carettoni, L., di Paola, S.: Http parameter pollution. OWASP AppSec Europe (2009)
25. Costello, A.: Rfc 3492: Punycode: A bootstring encoding of unicode for internationalized domain names in applications (idna) (2003)
26. Dhamija, R., Tygar, J.D., Hearst, M.: Why phishing works. In: Proceedings of the SIGCHI conference on Human Factors in computing systems (2006)
27. Durumeric, Z., Adrian, D., Mirian, A., Bailey, M., Halderman, J.A.: A search engine backed by Internet-wide scanning. In: 22nd ACM Conference on Computer and Communications Security (oct 2015)
28. Felt, A.P., Ainslie, A., Reeder, R.W., Consolvo, S., Thyagaraja, S., Bettes, A., Harris, H., Grimes, J.: Improving ssl warnings: Comprehension and adherence. In: Proceedings of the 33rd annual ACM conference on human factors in computing systems. pp. 2893–2902 (2015)
29. Felt, A.P., Reeder, R.W., Ainslie, A., Harris, H., Walker, M., Thompson, C., Acer, M.E., Morant, E., Consolvo, S.: Rethinking connection security indicators. In: Twelfth Symposium on Usable Privacy and Security ({SOUPS} 2016) (2016)
30. Felt, A.P., Reeder, R.W., Almuhimedi, H., Consolvo, S.: Experimenting at scale with google chrome's ssl warning. In: Proceedings of the SIGCHI conference on human factors in computing systems (2014)
31. Fielding, R.: Rfc 1808: Relative uniform resource locators (1995)
32. Hinden, R., Carpenter, B., Masinter, L.: Format for literal ipv6 addresses in url's. Tech. rep., RFC 2732, December (1999)
33. Jain, A.K., Gupta, B.: Phish-safe: Url features-based phishing detection system using machine learning. In: Cyber Security. Springer (2018)
34. Kettle, J.: Practical web cache poisoning. Port Swigger (2018)
35. Kreymer, I., Chuang, G.: Announcing the common crawl index! (2015)
36. Kumaraguru, P., Cranshaw, J., Acquisti, A., Cranor, L., Hong, J., Blair, M.A., Pham, T.: School of phish: a real-world evaluation of anti-phishing training. In: Proceedings of the 5th Symposium on Usable Privacy and Security (2009)
37. Kumaraguru, P., Rhee, Y., Sheng, S., Hasan, S., Acquisti, A., Cranor, L.F., Hong, J.: Getting users to pay attention to anti-phishing education: evaluation of retention and transfer. In: Proceedings of the anti-phishing working groups 2nd annual eCrime researchers summit (2007)
38. Kumaraguru, P., Sheng, S., Acquisti, A., Cranor, L.F., Hong, J.: Teaching johnny not to fall for phish. ACM Transactions on Internet Technology (TOIT) (2010)
39. Kunze, J.: Rfc 1736: Functional recommendations for internet resource locators (1995)
40. Le, A., Markopoulou, A., Faloutsos, M.: Phishdef: Url names say it all. In: IEEE INFOCOM. IEEE
41. Leitschuh, J.: Ssrf via maliciously crafted url due to host confusion. Hackerone (2019)
42. Ma, Z., Reynolds, J., Dickinson, J., Wang, K., Judd, T., Barnes, J.D., Mason, J., Bailey, M.: The impact of secure transport protocols on phishing efficacy. In: 12th {USENIX} Workshop on Cyber Security Experimentation and Test ({CSET} 19) (2019)
43. Marchal, S., Armano, G., Gröndahl, T., Saari, K., Singh, N., Asokan, N.: Off-the-hook: An efficient and usable client-side phishing prevention application. IEEE Transactions on Computers (2017)
44. Marlinspike, M.: More tricks for defeating ssl in practice. Black Hat USA (2009)
45. Muñoz, F.: Invalid url parsing with #. CVE-2016-8624 (2016)

46. Parekh, S., Parikh, D., Kotak, S., Sankhe, S.: A new method for detection of phishing websites: Url detection. In: 2018 Second international conference on inventive communication and computational technologies (ICICCT). IEEE (2018)
47. Postel, J.: Rfc: 761 ien: 129 (1980)
48. Reynolds, J., Kumar, D., Ma, Z., Subramanian, R., Wu, M., Shelton, M., Mason, J., Stark, E., Bailey, M.: Measuring identity confusion with uniform resource locators. Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (2020)
49. Sahingoz, O.K., Buber, E., Demir, O., Diri, B.: Machine learning based phishing detection from urls. Expert Systems with Applications (2019)
50. Sheng, S., Magnien, B., Kumaraguru, P., Acquisti, A., Cranor, L.F., Hong, J., Nunge, E.: Anti-phishing phil: the design and evaluation of a game that teaches people not to fall for phish. In: Proceedings of the 3rd symposium on Usable privacy and security (2007)
51. Thomas, K., Grier, C., Ma, J., Paxson, V., Song, D.: Design and evaluation of a real-time url spam filtering service. In: 2011 IEEE symposium on security and privacy. IEEE (2011)
52. Thompson, C., Shelton, M., Stark, E., Walker, M., Schechter, E., Felt, A.P.: The web's identity crisis: understanding the effectiveness of website identity indicators. In: 28th {USENIX} Security Symposium ({USENIX} Security 19) (2019)
53. Thompson, M.: The harmful consequences of the robustness principle (draft) (2018)
54. Tom: Security: uxss in chrome on ios. bugs.chromium.org (2018)
55. Tsai, O.: A new era of ssrf - exploiting url parser in trending programming languages! Black Hat USA (2017)
56. Tsai, O.: Breaking parser logic! take your path normalization off and pop 0days out. Black Hat USA (2018)
57. Vishwanath, A., Herath, T., Chen, R., Wang, J., Rao, H.R.: Why do people get phished? testing individual differences in phishing vulnerability within an integrated, information processing model. Decision Support Systems (2011)
58. Wang, X., Lau, W.C., Yang, R., Shi, S.: Make redirection evil again: Url parser issues in oauth. Black Hat Asia (2019)
59. Weber, C.: `https://websec.github.io/unicode-security-guide/` (2022)
60. Zouina, M., Outtaj, B.: A novel lightweight url phishing detection system using svm and similarity index. Human-centric Computing and Information Sciences (2017)