yoU aRe a Liar://A Unified Framework for Cross-Testing URL Parsers

Dashmeet Kaur Ajmani, Igibek Koishybayev, Alexandros Kapravelos
Department of Computer Science
North Carolina State University
{dmajmani,ikoishy,akaprav}@ncsu.edu

***Abstract*—A variety of attacks, including phishing, remote-code execution, server-side request forgery, and hostname redirection, are delivered to users over the web. The beginning of most of the web exploits is an *innocent-looking* URL. Malformed or misinterpreted URLs can lead to remote code execution attacks as well. The IETF and WHATWG standards organizations define the components of a URL and act as an implementation guide for URL parsers. They state which characters are allowed in each portion of the URL and loosely suggest what to do in case an undefined character is present in the URL. The existence of two standards is the first concern, and the addition of server-side request forgery in the latest version of OWASP Top 10, suggests that neither of these standards is being followed accurately and concisely. Moreover, neither of these specifications describe an exact implementation standard, causing inconsistencies in the way the various parsers interpret the same URL. For example, malicious users can find ways to craft URLs to look like they are pointing to one resource but actually direct the user to different one. This problem is worsened when one application uses two separate parsers for validation and resource fetching.**

In this paper, we design a framework that unifies the testing suites of 8 URL parsers from popular web-related projects and highlights the inconsistencies between them. We examine and dive deep into the URL parser implementation across the most popular libraries, browsers, and command-line tools, and discover many open areas for exploitation. Our findings include identifying categories of inconsistencies, developing proof-of-concept exploits, and highlighting the need for a comprehensive implementation standard to be developed and enforced at the earliest.

***Index Terms*—URL, parser, spoofing, web security, SSRF**

I. Introduction

To make it possible for an individual to record the location of a document, the World Wide Web associates each page of information with a unique identifier. The identifier consists of a string of characters that can be recorded in a computer file, written on a piece of paper, or sent to another person. An identifier used to specify a particular page of web information is called a *Uniform Resource Locator (URL)*. When a browser displays a page of information, it also displays the URL for the page. In other words, URLs are the standardized names for the Internet's resources. They are the resource locations that the browser needs to find pieces of electronic information. URLs are the first human access point to the Internet: a user points a browser at a URL, the browser sends the appropriate protocol messages to get the resource that is requested. These are a subset of a more general class of resource identifier called a *Uniform resource identifier (URI)*.

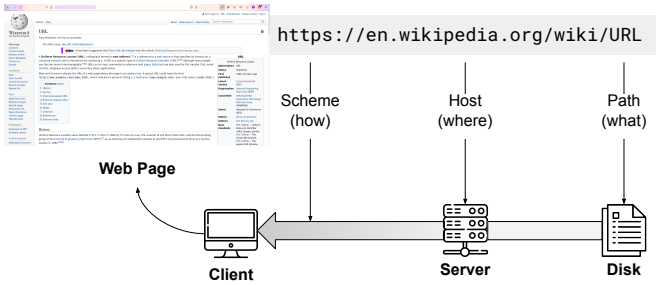

Fig. 1. How URLs relate to browser, machine, server, and location on the server's file-system

A URL may seem like a nonsensical collection of letters and punctuation. However, the precise syntax conveys meaning that a browser can use to retrieve a particular page, without passing through other documents. Each URL uniformly identifies a unique page of information by giving the name of a remote computer, a server on that computer, and a specific page of information available from that server. RFC 1738 [1] specifies the syntax and semantics of URLs, which is updated by RFC 3968 [2] specifying the detailed syntax of the super-set URIs. Figure 1 illustrates how the URL encodes the information. The initial part of the URL specifies an *access protocol* that tells the browser how to contact the remote server. The domain name of the server on which the server runs follows the colon and two slashes. Finally, a slash is used to separate the computer name from the suffix that identifies the specific item.

Although typical URLs have the same *scheme://server/location/path* structure as shown in the example in Figure 1, not all URLs follow the same form. Moreover, although different libraries and tools parse URLs accurately, it is plausible for the same URL to be parsed differently by different tools. For example, many browsers allow users to omit the protocol prefix; if no prefix is given, the browser adds *http://* to the URL. Such difference and confusion in URL parsing can cause unexpected behavior in the software or application, and could be exploited by threat actors to cause denial-of-service conditions, information leaks, host redirection, cache poisoning, or possibly conduct remote code execution attacks. As a matter of fact, the Open Web

Application Security Project (OWASP) added *Server-side Request Forgery (SSRF)* into the 10^{th} place of its top web application vulnerabilities in 2021 [3]. SSRF mostly occurs if the application is not handling a URL correctly. One reason for incorrect handling of a URL is different ways of parsing the URL with various tools and libraries. In this paper, we measure how different libraries, tools, and applications parse URLs and answer the following questions:

- Are there any differences in parsing the URLs by various tools and libraries?
- If there are differences, are they security relevant?
- Are there libraries that follow RFC specification or the WHATWG standard completely regarding parsing the URLs?

Contributions: The following are our main contributions:

1) We developed a framework that given a URL (or a list of URLs), outputs how each of the supported parsers would parse the URL(s) into its components. The framework also compares any pair of supported parsers, prioritizing observed differences based on the *same-origin-policy* (SOP) and other pre-determined rules.
2) We tested 8 URL parsers from different languages, browsers, and tools written in 5 programming languages. In total, we detected 4,262 inconsistencies with how each of the tools parse the same URL to its basic components.
3) We identify and categorize the most frequent parser inconsistencies into 7 categories.
4) In the spirit of open-source, we have released our framework code here: https://github.com/wspr-ncsu/urlparsing-framework.

II. Related Work

A. Academic Research

Reynolds *et. al.* [4] studied how well users can correctly determine the host identity of real URLs from common services and obfuscated "look-alike" URLs. They found that only 40% of obfuscated URLs were identified correctly, highlighting several ways in which URLs were confusing to users. Ahmed *et. al.* [5], and later Anitha *et. al.* [6], proposed a detection technique of phishing websites based on checking URLs of web pages. Several features were extracted from the URLs such as presence of "-" in the domain name, redirecting to another site using the symbol "//", and URLs having the presence of symbol "@". These works looked at URL parsers but in the context of phishing. Qilang *et. al.* [7] tested major browsers and URL scanners to expose the differences on the way they parsing URLs. In the process they found discrepancies in the way the browsers and scanners parse URLs. In this work, we dig deeper into the parsers themselves and propose a framework which will cross-evaluate parsers to reveal inconsistencies between them. There has been no other academic research that we are aware of on URL parsers, which proves our discoveries and evaluations to be novel.

B. Blogs and Conferences

In 2016, Daniel Stenberg, the founder and lead developer of curl wrote a blog [8] wherein he stated a couple of examples to point out that *"There's no unified URL standard and there's no work in progress towards that"*. Following that, in 2017, he published another blog [9] in which he highlighted the differences between the two specifications and urged the community to converge on a single URL standard. This work takes this one step further and shows the exact inconsistencies between parsers, and thus the standards, through a framework. Wang *et. al.* [10] combined URL parser issues with OAuth redirection mechanism and discovered new exploitation techniques for code/token stealing attacks and identified vulnerabilities in several popular OAuth providers. While they exploited some of the vulnerabilities arising from incorrect URL parsing, their focus was not examining the parser implementation. The most recent work shining light on how much impact incorrect URL parsing can have is by Orange Tsai in his BlackHat talk in 2017 [11]. In this talk he shows how *Server Side Request Forgery* attacks can be performed by taking advantage of the inconsistencies between URL parsers. Following this, in 2017 and in 2018, Orange published blogs about how he used this URL parser vulnerability, particularly abusing URL path parameter, to performing Remote Code Execution on GitHub Enterprise [12] and Amazon [13]. This work was inspired by Orange's contribution to SSRF, and by the astonishing fact that there has been no prior academic research conducted in this domain.

C. Concurrent Work

At the time of writing this paper, we discovered a report [14] published online in January 2022 by an industrial cybersecurity company called Claroty. In this report, the authors performed similar research and manually examined 16 URL parsers. While our work focuses on discovering inconsistencies and thus vulnerabilities using a framework, their work only consists of manually finding vulnerabilities. The framework that we introduce, the depth of tests, as well as the corresponding analysis that we perform using the framework, are things that increase the credibility and impact of this work. It builds a foundation for discovering more vulnerabilities in the future. This framework is designed with scalability in mind so that more libraries and tools can seamlessly be added in the future. The database of well-crafted URLs will also increase correspondingly.

III. Background

In this section, we describe URL components and the standards that define URLs.

A. Uniform Resource Locator

Even though we use URLs whenever we try to navigate to a website (http://), connect to a remote server (ssh://), transfer files between machines (ftp://), or connect to a database (mongodb://), we do not usually understand all parts of the

URL and its intricacies. In this subsection, we briefly describe the components of a URL.

In general, URL consists of five different components that are defined by RFC-3986 that are shown in Figure III-A. The exception to this rule is only when the URL is dealing with special schemes such as `javascript://`, `file://` and others. At first, it may seem that parsing URLs into components is not difficult. However, there are many challenges: the existence of two different standards, the lack of an exact algorithm on how to parse the URL in the initial versions of the RFC standard, the ability to use the same character sets in multiple URL components, the usage of identical control sequence, eg. a colon (:) to separate components, and backward compatibility issues. These challenges complicate the development of parsers that can deal with the many types of URLs.

The scheme is a protocol (algorithm) that should be used to deal with the URL. It is specific to very specific to the application.

The authority component of the URL consists of three parts: user info, host, and port. User info is further divided into username and password. Even though passing the password in URL is deprecated in RFC-3986 due to security risks, it is still widely used by other applications [15].

Web browsers implement Same-Origin-Policy to protect data available on a website when accessed by another website. Same-Origin-Policy, in short SOP, depends on the scheme, host, and port components of the URL to determine if the two different resources are from the same origin or not. Therefore, incorrect parsing of URLs, especially in browsers, can have a very significant impact on the entire security of the web.

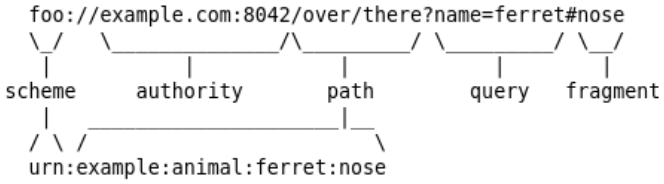

Fig. 2. Examples of two URIs and their component parts. Ref: RFC 3968

B. WHATWG vs RFC

Currently, there exist two URL specifications, RFC 3986 [2] by *Network Working Group* and WHATWG (*Web Hypertext Application Technology Working Group*) [16], which try to normalize how URLs should be interpreted. The existence of two different standards, that sometimes conflict with each other, further complicates the issue of correctly parsing the URLs. Not only is it hard to follow both standards at the same time, but it is also hard to follow a single standard thoroughly.

The RFC was developed before WHATWG and passed several changes in its lifetime. There are six revisions (RFC-1738, RFC-1808, RFC-2141, RFC-2396, RFC-2732, RFC-3986) of URLs in the RFCs, the latest of which, RFC-3986, was developed in 2005. Despite this, developers still use deprecated features due to backward compatibility issues as discussed in the previous subsection. On the other hand, WHATWG came out first in 2012 and is constantly evolving to include more features, edge cases, etc. This specification aims to solidify URL parsing, aligning multiple RFCs, and obsoleting them in the process. Most browsers today follow the WHATWG specification while other libraries and command-line tools follow RFC 3968 to implement their URL parsers.

C. Server-side Request Forgery

Incorrect parsing of the URL can enable different types of attacks. The most prominent of them is Server-Side Request Forgery, in short SSRF, which was included in 2021 into OWASP TOP-10 [3]. SSRF is a type of attack that exploits the trust between servers. For example, a backend server can assume that every request that is coming from the application server is trusted, and will thus serve some confidential information. Usually, developers employ allow-lists to protect applications from SSRF. However, this defense can be circumvented if the parts of the code base uses different parsers for URL parsing, *e.g.,* the validation section of the code parses URLs differently than the code that issues the request. In this case, the attacker can issue unauthorized requests by crafting a malicious URL [17], [18].

IV. Experimental Methodology

A. Identifying URL parsers

We crawled through the CWE and CVE databases [19]–[26] to identify the types of vulnerabilities existing among URL parsers. Using this information and based upon GitHub repository maintenance, contributors, and users, we identified the most popular URL parser libraries across programming languages. Table II in Appendix A shows a list of parsers and tools we used to test for URL parsing inconsistencies. We retrieved URL inputs from test suites of these libraries/tools in a semi-automated fashion by hooking them into the test framework code. Next, we crawled through the GitHub issues for those libraries, and 3 of the most popular browsers - Chrome, Mozilla, and Safari. Thereafter, from that list of vulnerabilities, as shown in Section V, we identified the most exploitable categories of these URL parsers, eg. insufficient URL sanitization, incorrect handling of confusable characters, incorrect handling of scheme, insufficient encoding, etc. Finally, we shortlisted the parsers we could extensively test and explore, based on their code being publicly available. [27]–[37]

B. The Framework

To test the parsers thoroughly, we needed a large corpus of URLs. We developed a framework that extracts this corpus from parsers test-suites, performs cross-testing of parsers, and reports on discovered bugs and inconsistencies. The framework performs the two main steps as shown in Figure IV-A.

Test Extraction. Since all of the parsers are widely used, we suspect they must be extensively tested. We hooked into test suites of each of the parsers by modifying a few code

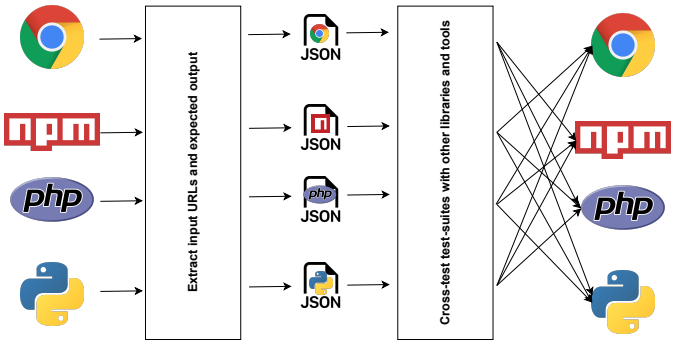

Fig. 3. Figure shows high-level architecture of our framework. The framework consists of mainly two parts, which are: (1) semi-automated pre-process step that generates uniform JSON files from existing test suites; (2) cross-testing part, which feeds inputs from JSON file into tools and detects differences from expected output

lines. Then the framework runs the respective parser's test-suites [38]–[47], and extracts the test URLs along with the parsed components into a JSON file. The framework stores the URLs from each test suite in individual files as well as in one large JSON file to create a database of URLs. Since testing suites usually contain all types of malicious and safe URLs, covering all the previous CVEs reported for their parser, we assume that this database is quite exhaustive.

Cross Testing. The next step is to cross-test the different parsers. This means that the input URL in the JSON file of one parser is tested using another parser. The outputs of both the parsers are then compared to find inconsistencies. These differences are prioritized based on pre-determined rules as listed in Appendix A. The highest priority is given to inconsistencies in the *same-origin-policy*, which is tuple of *(scheme, hostname, port)*. We use these prioritized differences for further analysis (Section V).

C. Analysis

For each parser, we analyze the differences to find potential security issues. We categorize the security issues into 7 number of categories as described in Section V. Finally, we shortlist a set of URLs that are most inconsistently parsed across parsers to create a *URLs of Interest* database. This shortlist is created to reduce the number of similar URLs as well as to document a set of URLs most likely to be abused by an adversary. Section V gives examples of some of these URLs. The framework is also used to check inconsistencies and inaccuracies between parsers and specifications.

V. Results

A. Statistics

As discussed in the previous section, we examined a total of 8 parsers, written in different programming languages. We extracted 1,445 URLs from their testing suites which we used in our dataset during cross-testing. The results of the cross-tests are highlighted in Table I. After analyzing the inconsistent behavior, we classified the differences into seven categories (see below). The framework discovered 4,262 inconsistencies among the parsers, of which 56% were differences in the SOP. These details are available in Table I. One observation is that `whatwg-url` is more restrictive compared to other parsers. Therefore, it throws an error whenever the URL is incorrect, *e.g.,* misses scheme, instead of relying on the default value.

B. Test Results

Using the framework, we found hundreds of inconsistencies among the examined parsers. After analyzing these inconsistencies, we categorized them into 7 categories, which can be viewed as the root causes of these differences. Using the categorizations, we can exploit parsers and combinations of parsers to cause unpredictable behavior, resulting in a wide range of vulnerabilities.

Since the URL syntax is complex, many edge-cases can be introduced to cause misinterpretation and malicious behavior. Sometimes, the specifications fail to clearly define how to handle the edge cases. Other times, there are discrepancies between the two specifications. We further describe the categories that the inconsistencies arise from due to the mishandled edge cases.

1) Hostname confusion: This type of inconsistency occurs when URL parsers set the hostname component for the same URL to different values. For example,

URL	http:\\\\a\\b:c\\d@foo.com\\
Chrome, whatwg-url, url-parse	host: a
cURL	host: foo.com
Urllib, urllib3, PHP, uri-js	path: \\\\a\\b:c\\d@foo.com\\

Another classic example that enables a redirect attack is shown in the table below. Despite CVE-2020-26291 [48] being reported an fixed on Node.js' uri-js library [29], we were surprised to see that many other parsers still haven't fixed this vulnerability in their parsers, including other popular Node.js libraries.

URL	http://google.com:80\\@yahoo.com
Chrome, urllib3, url-parse, whatwg-url	host: google.com
Urllib, uri-js, PHP, cURL	host: yahoo.com

These hostname inconsistencies can lead to hostname spoofing attacks, URL redirection attacks [20], and SSRF attacks [19].

2) Scheme confusion: RFC 3986 [2] mandates the presence of a scheme in a URL for it to be valid. When a scheme is not mentioned in the URL, some parsers assume a default scheme HTTP, while others consider the URL as a path, which is correct as per the RFC standard. For example,

URL	foo/bar
Chrome, urllib, cURL	host: foo
Urllib, PHP, url-parse, uri-js	path: foo/bar
Whatwg-url	Error

When a valid (well-known) scheme is not present, and there is a colon in the URL, some parsers assume everything up until

	Inputs from following tools / libraries. (Differences / Identical / Errors)						
	chrome (162)	**php (124)**	**urllib (164)**	**urllib3 (150)**	**url-parse (136)**	**uri-js (212)**	**whatwg-url (497)**
chrome		60 / 64 / 0	131 / 33 / 0	91 / 59 / 0	112 / 24 / 0	69 / 143 / 0	440 / 57 / 0
php	76 / 86 / 24		88 / 76 / 4	81 / 69 / 4	64 / 72 6	87 / 125 / 3	405 / 92 / 0
urllib	77 85 / 0	29 / 95 / 0		81 / 69 / 0	82 / 54 / 0	76 / 136 / 0	413 / 84 / 0
urllib3	97 / 65 / 24	53 / 71 / 8	104 / 60 / 5		93 / 43 / 0	123 / 89 / 0	407 / 90 / 0
url-parse	64 / 60 / 0	64 / 98 / 0	94 / 70 / 0	80 / 70 / 2		95 / 117 / 0	290 / 207 / 0
uri-js	33 / 91 / 0	76 / 86 / 0	66 / 98 / 0	40 / 110 / 0	62 / 74 / 0		320 / 177 / 0
whatwg-url	80 / 44 / 48	82 / 80 / 58	108 / 56 / 84	88 / 62 / 35	41 / 95 / 32	144 / 71 / 75	

TABLE I

THE TABLE SHOWS DIFFERENCES IN PARSING URLS IN CROSS TESTING. THE COLUMN SHOWS CORRESPONDING TEST FROM TOOL, WHILE THE ROW SHOWS THE NAME OF THE LIBRARIES THAT ARE TESTED. NOTE THAT THE NUMBER OF DIFFERENCES ALSO INCLUDE NUMBER OF TIMES THE PARSE THREW ERROR AND FAILED TO PARSE THE URL

the colon is the scheme. Other parsers exhibit unpredictable behavior. For example,

URL	www.php.net:80/index.php?test=1
Urllib, url-parse, uri-js, whatwg-url	scheme: www.php.net, host: empty
Urllib3, PHP, cURL	scheme: empty, host: www.php.net

Adversaries can abuse this inconsistency to bypass validation checks against certain hosts [49].

3) Control Characters confusion: The specification does not allow control characters like backspace, vertical tab, horizontal tab, line feed, etc in the URL. If these characters are present in the URL, they must always be percentage-encoded. However, a variety of peculiar behavior is observed among URL parsers. As displayed in the examples below, Chrome and Node.js' url-parse interpret the control characters as they are. Python's urllib3 library and Node.js' uri-js library encode them as expected. On the other hand, Python's urllib library and Node.js' whatwg-url library omit the control characters altogether. cURL throws a parsing error, which is also acceptable. PHP behaves in a special way, wherein it replaces the control characters with an underscore (_)!

URL	http://127.0.0.\r\n1:6379?SET\r\ntest\r\nfailure12:80
Chrome, url-parse	host: 127.0.0.\r\n1
Urllib3, uri-js	host: 127.0.0.%0D%0A1
Urllib, whatwg-url	host: 127.0.0.1
PHP	host: 127.0.0.__1
cURL	Error

URL	https://user:pass@xdavidhu.me\test.corp.google.com
Chrome, url-parse	host: xdavidhu.me est.corp.google.com
Urllib3, uri-js	host: xdavidhu.me%09est.corp.google.com
Urllib, whatwg-url	host: xdavidhu.meest.corp.google.com
PHP	host: xdavidhu.me_est.corp.google.com
cURL	Error

Control character confusion can lead to CRLF injection attacks [50] and Header injection attacks [51]. For eg. [52].

4) Backslash confusion: According to RFC 3698, a backslash is different than a forward slash and should not be interpreted as one. However, some modern browsers and parsers use forward slash and backslash interchangeably, because the WHATWG URL specification states that they should be treated in the same way. This is opposite to the RFC which suggests percentage encoding of non-reserved characters. As shown through the below example, Chrome, whatwg-url, and url-parse libraries omit backslashes or convert them to forward slashes and assume the URL to be in a valid form. Urllib3, uri-js encode the backlashes, while PHP and cURL exhibit unpredictable behavior.

URL	https:/\\/\\/\\github.com/foo/bar
Chrome, whatwg-url	host: github.com
Urllib3, uri-js	path: /%5C/%5C/%5Cgithub.com/foo/bar
Urllib	path: /\\/\\/\\github.com/foo/bar
PHP	path: /\/\/\github.com/foo/bar
cURL	host: \

Exploiting this category can enable a malicious attacker to easily bypass many different validations. It can also lead to Hostname direction attacks, eg. URL: "http://google.com:80\\@yahoo.com/#what\\is going on".

5) Slash confusion: The addition of a non-standard number of slashes, especially in the authority section of the URL, causes inconsistencies in authority detection. We found that modern browsers omit or ignore the extra slashes, and treat the URL as valid. Some URL parsers behave according to RFC 3986, which states that the authority is followed by a colon and a double-slash until the end-of-line or a delimiter is encountered. More than two slashes cause parsers to interpret the entire URL as a path. For example,

URL	foo:////////bar.com/
Chrome	host: bar.com
Urllib, urllib3, uri-js, url-parse, whatwg-url	path: ///////bar.com/
PHP	Error

cURL is optimized for usability and its behavior depends on the number of slashes. This is clearly stated in cURL's documentation [53].

URL	cURL's Behaviour
http:/google.com	host: google.com
http:///google.com	host: google.com
http:////google.com	Error

An adversary might be able to bypass security checks because of this confusion and can cause a variety of exploits. Sub-section VI-A describes one proof-of-concept exploit.

6) Path confusion: Inconsistencies in parsing double-dots in the path segment of the URL can cause serious vulnerabilities. We found that some URL parsers discard the dots and assume an absolute path, while others retain the dots. This is another instance when cURL misbehaves!

URL	../g
Chrome, urllib	path: /g
Urllib, uri-js, url-parse, php	path: /g
Whatwg-url	Error
cURL	host: ..

This vulnerability can lead to information leakage attacks via directory traversal [54].

7) Encoding confusion: This category is for URLs that contain encoded characters. RFC 3986 states that all URL components except the scheme can be represented using URL encoded characters, and they should be decoded when parsed. However, some parsers do not encode non-ASCII characters even though the specification suggests doing so. *"Non-ASCII characters must first be encoded according to UTF-8 [STD63], and then each octet of the corresponding UTF-8 sequence must be percent-encoded to be represented as URI characters."*

As shown in the example below, chrome and url-parse decode the Unicode characters. Whatwg-url, uri-js, and urllib3 perform IDNA encoding [55], i.e. convert Unicode characters in the host to their ASCII equivalent representation. The "xn–" says "everything that follows is encoded-Unicode." On the other hand, urllib, PHP, and cURL do not decode the Unicode characters.

URL	http://\u30d2:\u30ad@\u30d2.abc.\u30cb/\u30d2
Chrome, url-parse	host: ヒ.abc.ニ
Urllib3, uri-js, whatwg-url	host: xn–pdk.abc.xn–idk
Urllib, PHP, cURL	host: \u30d2.abc.\u30cb

An adversary could bypass validations by decoding the URL she wants to retrieve. This inconsistency can cause URL misinterpretation and URL look-alike attacks [56].

VI. Exploiting URL parsers

We show two concrete ways in which we combine various URL parser inconsistencies discussed in Section V to break web security. The basic attacks depend on the usage of multiple URL parsers that behave inconsistently. Although we show only two PoCs, we emphasize that a vast range of vulnerabilities can be exploited using the similar ideology. The code listings are shown in Appendix A.

A. Exploiting Backslash/Hostname Confusion

Most real-world applications have a huge code-base, modularized into multiple files, folders, and even applications in the case of a distributed application. Along with that, application developers keep changing and rotating over time. In such organizations, it's often difficult to maintain consistent usage of third-party libraries across modules. In this PoC attack, we assume two separate modules - one is a utility module (Listing 1) which checks if a given URL is valid using `urllib3`, and the other module implements a file download feature using `requests` which depends on `urllib` (Listing 2). By passing specially crafted URL, *e.g.,* `http://example.com:80@localhost:8080/secret.txt`, an attacker can pass the checks and read `secret.txt` from localhost.

B. Exploiting Slash Confusion

For this exploit, consider a PHP application (Listing 3) that, by design, has a feature vulnerable to SSRF. To mitigate this vulnerability, the application validates URLs using a blacklist of hostnames. If a match is found, it blocks the requests to the host. Otherwise, it passes the URL into cURL to fetch the resource. If the URL parser used to validate the URLs follows RFC-3968, it will return an empty host. And if the tool used to fetch the resource is one that simply ignores multiple slashes, it'll go to the malicious website. For example, a call to `https:///evil.com` will bypass the validation check.

VII. Limitations and Future Work

Although we tried to expand our research to a variety of URL parser tools, across a handful of programming languages, we still examined a limited number of URL parser libraries. The framework is not fully automated and still requires manual effort during URL extraction from parsers test-suite to hook into test-code. Another limitation is that the exploits are only proof-of-concept attacks and not real-world attacks. Chrome canonicalizes URLs [57] wherein it transforms specific URL components or specific types of URLs into a standard form. This detaches the URL parsing results from the actual browser results (i.e., we found URLs that Chrome's library did not parse correctly, but when we tested them in the browser, they were corrected).

In future work, we plan to look into popular web frameworks such as WordPress, Express.js, and others to detect URL parsing inconsistencies. We also plan to improve on the automation of the framework. Since we have identified categories of the discovered inconsistencies, we plan on automating the categorization of inconsistencies in the framework itself. The framework should be able to perform a more detailed analysis. We want to modularize and expand the framework to support more parsers.

VIII. Conclusion

The procedure of developing a precise and exhaustive *URL syntax and implementation* specification, or enforcing this specification, is an extremely challenging task. Currently, there exist two URL standards and the URL parsing libraries, browsers, or tools do not entirely follow either of the standards. Due to the complex nature of URLs, the standards also miss a lot of corner cases which leads to discrepancies between URL parsers and open them up to exploitation. To make matters worse, differences are observed across libraries in the same programming language. In this work, we tested 8 URL parsers including 1 browser, 1 command-line tool, and 6 programming language libraries.

The framework we developed first generated a URL database consisting of 1,445 URLs fetched from the testing suites of all the examined parsers. It then cross-tested all the parsers with this URL database exposing 4,262 inconsistencies in their parsed outputs. We performed an extensive analysis of these inconsistencies and identified 7 categories of URL parser vulnerabilities.

As far as we are aware, this is the first academic study diving into the depths of URL parsers. Understanding the existence of inconsistencies between URL parsers, and gaining detailed knowledge on how a specific parser behaves when fed with edge cases, will help developers pick an appropriate URL parser for their application. The lack of a prior study, and the rise in the number of SSRF exploits, suggest an urgent need to look into the standardization and enforcement of rules to harden the first layer of defense in the web security world.

Acknowledgements. We thank the anonymous reviewers for their helpful feedback. This work was supported by the National Science Foundation under grant CNS-2047260 and by the Office of Naval Research under grant N00014-21-1-2159.

APPENDIX

List of parsers and tools we used to test for URL parser inconsistencies, their popularity statistics, and the number of URL parsing CVEs reported on each of the parsers.

Environment	Name	Downloads	CVE count	Standard
npm	whatwg-url	40.9M / week	2	WHATWG
npm	uri-js	30M / week	2	RFC-3968
npm	url-parse	13.7M / week	4	WHATWG
php	parse_url	77.9 % [1]	5	RFC-3968
python	urllib	43.1k stars	4	RFC-3968
python	urllib3	2.9k stars	3	RFC-3968
browser	chrome	63.58 % [2]	12	WHATWG
cli	*cURL*	23.8k stars	5	RFC-3968

TABLE II
THE LIST OF TOOLS AND LIBRARIES THAT WE USED TO TEST FOR URL PARSING INCONSISTENCIES

[1] Of all websites whose server-side programming language is known [58].
[2] Chrome's worldwide market share. [59].

An adversary can use a well-craft a URL to upload a malicious file or cause a user to download one.

```python
from urllib3.util.url import parse_url
def is_valid(url):
    # Verify url belongs to whitelisted domain and secure
    ↪  scheme + port
    parsed = parse_url(url)
    if parsed.scheme == 'https' and parsed.hostname ==
    ↪  'example.com' and parsed.port == 443:
        print("URL valid.")
        return True
    else:
        return False
```

Listing 1: The snippet verifies if the URL is in allowed list of URLs, and if so return True

```python
import requests
from urllib.parse import urlparse

url = sys.argv[1]
if util.is_valid(url):
  parsed = urlparse(url)
  remote = 'http://' + parsed.hostname + ':' +
  ↪  str(parsed.port) + parsed.path
  # Download file
  response = requests.get(remote)
  print("File downloaded from URL: ", response.url)
else:
  print("Invalid Request.")
```

Listing 2: The code takes as input a URL and prints the content of the file

The tool will will fetch a confidential resource because the parser that validates the host against an allow-list ignores multiple slashes.

```php
<?php
$url = "https:///evil.com";
$parsed = parse_url($url);
# Check if the URL is blacklisted
if ($parsed["host"] == "evil.com") {
    print("URL not allowed.\n");
    exit();
}
$curl = curl_init($url);
$data = curl_exec($curl);
print($data);
?>
```

Listing 3: The snippet checks if the URL is not blacklisted before passing it to cURL which ignores multiple slashes

Rules	Outcome	Examples
If the difference is cosmetic	Ignore	path: 'h' and path: '/h'. Urllib3 combines password and username as 'user:pass' and others separate them out. Square brackets around IPv6 hosts '[dead:beef::1]'.
If there is a letter-case difference in the hostname	Ignore	'GooGle.com' is converted to 'google.com'.
If there is a letter-case difference in the scheme	Ignore	For url 'abOut://eXamPlE.com?info=1', urllib3 interprets scheme as 'about' while php interprets it as 'abOut'. Same with 'HTTP'.
If a library does not parse an URL and returns an error	Retain	When port is not a valid integer 'http://www.example.net:foo', php and urllib3 return an error while urllib parses it as username.
If the difference is in encoding	Retain	URL '/rest/Users?filter={"id":"123"}' is encoded by urllib3 as 'filter=%7B%22id%22:%22123%22%7D'; not by php.
If there's a SOP or 'scheme, host, port' difference	Retain	If the scheme is empty, urllib by default assume it to be HTTP.

TABLE III
THE LIST OF PRE-DETERMINED RULES USED TO CLASSIFY AND PRIORITIZE THE PARSER INCONSISTENCIES.

REFERENCES

[1] IETF, "Uniform Resource Locators (URL)," https://datatracker.ietf.org/doc/html/rfc1738.
[2] ——, "Uniform Resource Identifier (URI): Generic Syntax," https://datatracker.ietf.org/doc/html/rfc3986.
[3] OWASP, "Server-side Request Forgery," https://owasp.org/Top10/A10_2021-Server-Side_Request_Forgery_(SSRF)/, 2021.
[4] J. Reynolds, D. Kumar, Z. Ma, R. Subramanian, M. Wu, M. Shelton, J. Mason, E. Stark, and M. Bailey, *Measuring Identity Confusion with Uniform Resource Locators*. Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3313831.3376298

[5] A. A. Ahmed and N. A. Abdullah, "Real time detection of phishing websites," in *2016 IEEE 7th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, 2016.

[6] A. Anitha, K. S. Gudivada, R. L. M.K, S. Kumari, and C. UshaCR, "Identifying phishing websites through url parsing," *International journal of engineering research and technology*, vol. 8, 2019.

[7] Q. Yang, D. Damopoulos, and G. Portokalidis, "Wysisnwiv: What you scan is not what i visit," in *RAID*, 2015.

[8] Daniel Stenberg, "My URL isn't your URL," https://daniel.haxx.se/blog/2016/05/11/my-url-isnt-your-url/, 2016.

[9] ——, "One URL standard please," https://daniel.haxx.se/blog/2017/01/30/one-url-standard-please/, 2017.

[10] Xianbo Wang, Wing Cheong Lau, Ronghai Yang, and Shangcheng Shi, "Make Redirection Evil Again: URL Parser Issues in OAuth," https://i.blackhat.com/asia-19/Fri-March-29/bh-asia-Wang-Make-Redirection-Evil-Again-wp.pdf, 2019.

[11] Orange Tsai, "A New Era of SSRF - Exploiting URL Parser in Trending Programming Languages," https://www.youtube.com/watch?v=voTHFdL9S2k, 2017.

[12] O. Tsai, "How I Chained 4 vulnerabilities on GitHub Enterprise, From SSRF Execution Chain to RCE!" https://blog.orange.tw/2017/07/how-i-chained-4-vulnerabilities-on.html, 2017.

[13] ——, "How I Chained 4 Bugs(Features?) into RCE on Amazon Collaboration System," https://blog.orange.tw/2018/08/how-i-chained-4-bugs-features-into-rce-on-amazon.html, 2018.

[14] Noam M., Sharon B. of Claroty Team82, Raul O., Kirill E. of Snyk, "Exploiting URL Parsers: The Good, Bad, and Inconsistent," https://claroty.com/wp-content/uploads/2022/01/Exploiting-URL-Parsing-Confusion.pdf, January 2022.

[15] MongoDB Documentation, "Connection String URI Format," https://docs.mongodb.com/manual/reference/connection-string/, Accessed: 22 Feb 2022.

[16] "WHATWG: URL Living Standard," https://url.spec.whatwg.org/, accessed: 23 Feb 2022.

[17] G. Niedziela, "Bypassing domain deny_list rule in smokescreen via trailing dot leads to ssrf," https://hackerone.com/reports/1410214, 2021.

[18] H. Jaiswal, "Ssrf chained to hit internal host leading to another ssrf which allows to read internal images," https://hackerone.com/reports/826097, 2020.

[19] CWE, "Cwe-918: Server-side request forgery (ssrf)," https://cwe.mitre.org/data/definitions/918.html.

[20] ——, "Cwe-601: Url redirection to untrusted site ('open redirect')," https://cwe.mitre.org/data/definitions/601.html.

[21] ——, "Cwe-84: Improper neutralization of encoded uri schemes in a web page," https://cwe.mitre.org/data/definitions/84.html.

[22] ——, "Cwe-233: Improper handling of parameters," https://cwe.mitre.org/data/definitions/233.html.

[23] ——, "Cwe-22: Improper limitation of a pathname to a restricted directory ('path traversal')," https://cwe.mitre.org/data/definitions/22.html.

[24] ——, "Cwe-939: Improper authorization in handler for custom url scheme," https://cwe.mitre.org/data/definitions/939.html.

[25] ——, "Cwe-472: External control of assumed-immutable web parameter," https://cwe.mitre.org/data/definitions/472.html.

[26] Mitre, "Cves on url parsers," https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=url+parse.

[27] Openbase, "10 Best Node.js URL Parsing Libraries," https://openbase.com/categories/js/best-nodejs-url-parsing-libraries.

[28] Unshift, "Node.js url-parse Library," https://github.com/unshiftio/url-parse.

[29] K. Cox, "Node.js uri.js Library," https://github.com/kevincox/url.js.

[30] Python, "Python urllib Library," https://github.com/python/cpython/blob/3.9/Lib/urllib/parse.py.

[31] URLlib3, "Python urllib3 Library," https://github.com/urllib3/urllib3.

[32] cURL, "C libcurl Library," https://github.com/curl/curl/blob/master/lib/url.c.

[33] GNU, "GNU Wget," https://git.savannah.gnu.org/cgit/wget.git/tree/src/url.c.

[34] PHP, "Php parse_url library," https://www.php.net/parse_url.

[35] NPM, "Npm whatwg-url library," https://www.npmjs.com/package/whatwg-url.

[36] Chromium, "Chrome's url library," https://source.chromium.org/chromium/chromium/src/+/master:url/.

[37] Mozilla, "Mozilla's url parser," https://hg.mozilla.org/mozilla-central/file/tip/netwerk/base/nsURLParsers.cpp.

[38] Unshift, "Node.js url-parse Testing Suite," https://github.com/unshiftio/url-parse/tree/master/test.

[39] K. Cox, "Node.js uri.js Testing Suite," https://github.com/garycourt/uri-js/tree/master/tests.

[40] Python, "Python urllib Testing Suite," https://github.com/python/cpython/tree/3.10/Lib/test.

[41] URLlib3, "Python urllib3 Testing Suite," https://github.com/urllib3/urllib3/tree/main/test.

[42] cURL, "C libcurl Testing Suite," https://github.com/curl/curl/blob/master/tests/libtest/lib1560.c.

[43] GNU, "GNU Wget Testing Suite," https://git.savannah.gnu.org/cgit/wget.git/tree/tests.

[44] PHP, "Php parse_url testing suite," https://github.com/php/php-src/tree/master/ext/standard/tests/url.

[45] NPM, "Npm whatwg-url testing suite," https://github.com/jsdom/whatwg-url/tree/master/test.

[46] Chromium, "Chrome's url testing suite," https://source.chromium.org/chromium/chromium/src/+/master:url/url_parse_unittest.cc.

[47] Mozilla, "Mozilla's url parser testing suite," https://hg.mozilla.org/mozilla-central/file/tip/browser/components/urlbar/tests/browser.

[48] CVE-2020-26291, "Hostname spoofing via backslashes in url." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-26291.

[49] CWE, "Cwe-288: Authentication bypass using an alternate path or channel," https://cwe.mitre.org/data/definitions/288.html.

[50] ——, "Cwe-93: Improper neutralization of crlf sequences ('crlf injection')," https://cwe.mitre.org/data/definitions/93.html.

[51] ——, "Cwe-113: Improper neutralization of crlf sequences in http headers ('http response splitting')," https://cwe.mitre.org/data/definitions/113.html.

[52] CVE-2020-11078, "Crlf injection in httplib2." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-11078.

[53] curl, "cURL docs: URL syntax and their use in curl," https://github.com/curl/curl/blob/c61ca433402876fefef1dab847acda1647b7dab9/docs/URL-SYNTAX.md, 2007.

[54] CWE, "Cwe-23: Relative path traversal," https://cwe.mitre.org/data/definitions/23.html.

[55] Network Working Group, "Internationalizing Domain Names in Applications (IDNA)," https://datatracker.ietf.org/doc/html/rfc3490, 2003.

[56] CWE, "Cwe-1007: Insufficient visual distinction of homoglyphs presented to user," https://cwe.mitre.org/data/definitions/1007.html.

[57] C. R. document, "Chrome's url canonicalization," https://chromium.googlesource.com/chromium/src/+/HEAD/url/README.md.

[58] W3 Techs, "Usage Statistics and Market Share of PHP for Websites," https://w3techs.com/technologies/details/pl-php, March 2022.

[59] Brian Dean, "Google Chrome Statistics for 2022," https://backlinko.com/chrome-users.

[60] A. Barth and C. Reis, "The security architecture of the chromium browser," 2009.

[61] M. Zalewski, *The Tangled Web*. No Starch Press, 2011.

[62] ——, *Browser Security Handbook*. Google Inc., 2010.

[63] A. Christensen, "URL Parsing in WebKit," https://webkit.org/blog/7086/url-parsing-in-webkit/, 2016.

[64] Google, "Google Safe Browsing," https://developers.google.com/safe-browsing.

[65] W. C. G. D. Report, "URLPattern API," https://webkit.org/blog/7086/url-parsing-in-webkit/, 2019.

[66] "WHATWG: URL GitHub Repository," https://url.spec.whatwg.org/, accessed: 23 Feb 2022.

[67] Orange Tsai, "Breaking Parser Logic: Take Your Path Normalization off and Pop 0days Out!" https://www.youtube.com/watch?v=CIhHpkybYsY, 2020.