dippy_gram: Grammar-Aware, Coverage-Guided Differential Fuzzing of URL Parsers

Ben Kallus and Sean W. Smith
Department of Computer Science
Dartmouth College, Hanover, NH, USA
{benjamin.p.kallus.gr,sws}@dartmouth.edu

May 3, 2023

Abstract

URL parsing confusion is a perennial source of security vulnerabilities in web services (e.g., [Mos+; Cha; WTA]). We present a new differential fuzzing framework, and demonstrate its effectiveness at discovering parsing bugs in popular URL parsers written in Python. Prior approaches to coverage-guided differential fuzzing examine only program exit codes to determine when a discrepancy has been encountered. We expand upon this technique by also examining program output for differentials. Our fuzzer, dippy_gram, has uncovered numerous bugs in URL parsers, including the parser in the Python standard library, and the parser in the most-downloaded Python package on PyPI.

1 Introduction

This paper presents *dippy_gram*, a **di**fferential fuzzer for **p**arsers, targeting **Py**thon and using **gram**mar-based mutations. Section 2 reviews the problem and the prior work. Section 3 presents our methodology and Section 4 presents our current results. Section 5 discusses the next steps, and Section 6 concludes.

2 Background and Prior Work

2.1 URL Standards

URL is defined by two different standards; IETF RFC 3986 [BFM] and its extensions [DS], and the WHATWG URL Living Standard [WHA].[1] There are many validation differences between the RFC standards and the WHATWG standard, such as whether port numbers outside the unsigned 16-bit range are permissible, but the primary distinction between the two standards is that the WHATWG standard aims to clearly define the handling of errors in malformed URLs, whereas the RFC leaves error handling to the implementer. While the IETF maintains that RFC 3986 is the current URL standard, the WHATWG standard states explicitly that it exists to obsolete the RFCs.

Thus, modern URL parsers thus fall into two camps: RFC-compliant and WHATWG-compliant, but many parsers also target partial compatibility with both standards, or backwards-compatibility with obsolete RFCs, leading to a diverse set of behaviors among "standards-compliant" parsers.

[1] Although RFC 3986 distinguishes between URIs and URLs, we follow the convention of the WHATWG standard and ignore that distinction.

```
scheme://user@host:port/path?query#fragment
```

Figure 1: The pieces of a URL in the familiar form.

2.2 URL Components

A URL is composed of at most seven string-valued components: *scheme*, *userinfo*, *host*, *port*, *path*, *query*, and *fragment*. (See Figure 1.) Their conventional purposes are as follows:

- A URL begins with a *scheme*, which specifies the protocol over which its resource is accessible. The *scheme* is the only mandatory component of a URL; all others may be empty.
- A URL's *userinfo* specifies the user who is to access the URL's resource. Although this field has also historically specified a password, this is not supported in modern URL standards for security reasons.
- A URL's *host* specifies the machine that hosts the URL's resource. Usually, this component is an IP address or a domain.
- A URL's *port* specifies the port over which the resource is accessible. This component is often omitted, since applications generally associate a default port with each scheme.
- A URL's *path* specifies where the resource resides on the host's filesystem.
- A URL's *query* and *fragment* components serve to augment the *path* with extra information.

The standards also specify other URL forms that eschew the "`//`" after the *scheme* and colon. These URL forms must also omit the *userinfo*, *host*, and *port* URL components, and thus consist of at most a *path*, *query*, and *fragment*. An example of a popular URL scheme that uses one of these forms is the `magnet` scheme, which specifies a resource by its hash value, and therefore needs only to utilize the *scheme* and *query* URL components.

Some valid URLs do not reference valid resources. For example, `file:///etc/passwd` is a valid URL, but probably does not identify a resource on a Windows system. Conversely, although "`example.com`" is not a valid URL because it has no *scheme*, it does clearly identify a resource and is thus often acceptable in place of a URL. Such identifiers are referred to alternately as *URL references* and *relative URLs*. In many cases, such as in browser address bars, it is useful to accept both absolute and relative URLs, but the union of their grammars is ambiguous, so one cannot reliably be distinguished from the other. For example, it is clear from context that `tel:911` references the North American emergency telephone number, and thus has *scheme* `tel` and *path* `911`, but it is similarly clear that `localhost:911` references access to port `911` on the host `localhost` without specifying a *scheme*.

2.3 Fuzzing

A fuzzer is a program that takes as input a target program T, then repeatedly runs T on generated inputs until T demonstrates undesirable behavior, such as crashing, accessing memory out of bounds, or exceeding a running time threshold (e.g. [Pet+17b]). The fuzzer then reports the offending input to the user.

2.4 Coverage-Guided Fuzzing

A *coverage-guided* fuzzer traces its target's execution on each input, and uses the execution information to determine which inputs should be selected for mutation. In each run of the target, a coverage-guided fuzzer extracts its target's execution trace, represented as a walk through the target's control flow graph. The execution trace of a program T on input I_1 can be thought of as a fingerprint of I_1. If another input I_2 leads to exactly the same execution path in T, then the reasoning goes that I_2 is sufficiently similar to I_1 that it is not worth mutating further. On the other hand, inputs that induce walks with interesting features are selected for further mutation. This might include walks that visit previously-unvisited vertices, edges, or edge sequences.

2.5 Grammar-Based Fuzzing

A *grammar-based* fuzzer uses the structure of its target's input language to generate fuzzing inputs. This

allows the fuzzer to generate inputs for formats with features that are impractical to generate randomly, such as checksums and magic numbers.

2.6 Differential Fuzzing

A *differential* fuzzer is a program that takes as input a set of n similar target programs, $T_1, \ldots, T_n$, then searches for inputs to the targets that cause their outputs to differ. This search might be unguided [Jab+21], it might examine coverage information [Pet+17a], or it might be guided by a constraint solver [Nol+20]. One coverage metric that can be used to guide a differential fuzzer is fine path δ-diversity [Pet+17a]. A differential fuzzer guided by fine path δ-diversity runs each input I on each of the n targets, and produces fingerprint$_I$, which is defined by having i^{th} entry equal to the set of edges traversed in T_i's execution trace on I. If an input's fingerprint has not been previously encountered, that input is selected for mutation. Thus, a differential fuzzer guided by fine path δ-diversity selects inputs that cause at least one target to exhibit new coverage in the context of its peer targets' coverage.

Algorithm 1 Differential Fuzz

```
1:  queue ← seeds
2:  explored ← []
3:  while queue is not empty do
4:      mutation_queue ← []
5:      for input in queue do
6:          trace_sets ← []
7:          statuses ← []
8:          stdouts ← []
9:          for parser in parsers do
10:             process ← run(parser, input)
11:             trace_sets.add(set(process.trace))
12:             statuses.add(process.exit_status)
13:             stdouts.add(process.stdout)
14:         fingerprint ← hash(trace_sets)
15:         if fingerprint not in explored then
16:             if statuses is neither all zero nor all nonzero
    then
17:                 report(input)
18:             else if stdouts are not all identical
    then
19:                 report(input)
20:             else
21:                 mutation_queue.add(input)
22:             explored.add(fingerprint)
23:     queue ← []
24:     while queue.length < generation_size do
25:         for input in mutation_queue do
26:             queue.add(mutate(input))
```

3 Methodology

dippy_gram is a generational fuzzer. Its seed generation is a collection of 200 URLs from WebKit's test suite. In each generation, each input is fed to each of the target parsers, and each parser's execution is traced using `afl-showmap`. Then, each input's collection of traces is hashed. If that hash has been previously encountered, the input is ignored. Otherwise, if a differential has been detected, through either exit statuses or differences in the targets' outputs, it is reported. Finally, if no differential is found, but the hash is still new, the URL is selected for mutation and propagates forward into the next generation.

Note that NEZHA[Pet+17a], reports all encountered differentials, whereas dippy_gram ignores differentials that provide no new δ-diversity. This choice was made because without this filtering, duplicate differentials clutter the fuzzer's output.

To determine whether two parsers exhibit differential behavior, dippy_gram compares both return

codes and serialized parser output. We term the situation in which a URL is accepted by at least one parser and and accepted by at least one other parser to be an *exit status differential*. We term the situation in which a URL is accepted by all parsers, but in which different parsers assign different values to different URL fields to be an *output differential*. Because both types of differentials require comparison to a baseline in order to recognize, they cannot be easily detected by single-target fuzzing.

When an input is selected for mutation, it is mutated by one of four mutation operations: random byte replacement, deletion, insertion, and parse subtree replacement. At each mutation step, a mutation is chosen at random from among those that are applicable. Similarly, an input that violates the fuzzer's internal representation of the URL grammar cannot be selected for parse subtree replacement. This presents a dilemma. If the fuzzer's ground truth URL grammar is too permissive, then grammar-mutated URLs do not exhibit sufficient structure. On the other hand, if the ground truth grammar is too rigid, then grammar-based mutations are applicable only in the event that an input is particularly well-formed. We implement the full RFC, and acknowledge that a better approach might implement a permissive superset of the RFC.

4 Results

We have run experiments with six URL parsers written in Python: urllib3, rfc3986, Hyperlink, yarl, furl, and CPython's urllib.

- urllib3 is an HTTP library with an included URL parser. Because it is a dependency of many popular Python packages, including Requests and AWS-CLI, urllib3 stands as the second-most-downloaded package on the Python package repository PyPI [PyP].
- rfc3986 is a URL parsing library that is a dependency of the popular HTTP library HTTPX.
- yarl is a URL parsing library that is a dependency of another popular HTTP library, AIOHTTP.
- Hyperlink is a URL parsing library that is a dependency of the Twisted network programming framework.
- furl is a somewhat less actively-developed URL parser that was selected for its comparable download statistics to Hyperlink.
- urllib is the URL module in the Python standard library. It is used widely in the Python ecosystem, including within the Django web framework

`.://example.com`

Parser	Scheme	Host	Path
CPython	`.`	`example.com`	
yarl	`.`	`example.com`	`/`
Hyperlink	`.`	`example.com`	`/`
furl			
rfc3986			`.://example.com`
urllib3		`.`	`//example.com`

Table 1: How the targeted parsers parse one example malformed URL.

We have reported numerous parsing bugs found by dippy_gram. Two have been patched in CPython, one has been patched in rfc3986, one has been patched in urllib3, and many others are under review.

The bugs fall into several categories, a few of which we discuss below. The examples given were hand-minimized to aid interpretation. Because URLs are often embedded into other protocols, injection into URLs of control characters for other protocols, such as CRLF for HTTP, can amount to singificant security vulnerabilities. Thus, even seemingly harmless URL parsing bugs may have security-related consequences in the right context.

4.1 Scheme Parsing Bugs

The URL standards define a scheme string to be an string of composed of ASCII alphanumeric characters, plus signs, minus signs, and periods in which the first character is alphabetical. Table 1 shows the result of parsing "`.://example.com`" with six popular Python URL parsers.

All six parsers accept the malformed URL without error. CPython, yarl, and Hyperlink make the relatively simple mistake of interpreting the period as

`e.vil://go.od`

Parser	Scheme	Host	Path
CPython	`e.vil`	`go.od`	
yarl	`e.vil`	`go.od`	`/`
Hyperlink	`e.vil`	`go.od`	`/`
furl	`e.vil`	`go.od`	
rfc3986	`e.vil`	`go.od`	
urllib3		`e.vil`	`//go.od`

Table 2: How the targeted parsers handle periods within schemes.

a scheme. furl, returns a URL object in which every field is empty. rfc3986 correctly recognizes that a scheme cannot begin with a period, and attempts to parse the input as a relative URL with a relative path beginning with a colon. This behavior is also in violation of the RFC because the first segment of a relative path in a relative URL is not permitted to contain a colon, presumably because of this ambiguity.

urllib3 does not permit periods within schemes at all, and thus interprets the period as the host, the colon as the port delimiter, and the first slash as the path delimiter. Because urllib3 does not allow for dotted schemes, it uniquely misinterprets the URL "`e.vil://go.od`," as can be seen in Table 2. This differential may have the potential for application to an open-redirect attack because of its ability to cause urllib3 to misinterpret the host.

4.2 Port Parsing Bugs

RFC 3986 defines a port to be a string of zero or more ASCII digits. The WHATWG standard defines a port string similarly, but stipulates that its value must not exceed 65535. Thus, an RFC-compliant parser is necessarily in violation of the WHATWG standard, and vice-versa.

urllib3, which claims RFC 3986 compliance, also prohibits ports greater than 65535. However, it does so with a regular expression that does not account for leading zeros. Thus, one can construct a URL that will be erroneously rejected by urllib3 by prefixing its port number with leading zeros such that the length of the port string is greater than five.

`http://example.com: +8_0`

Parser	Scheme	Host	Port	Path
CPython	http	`example.com`	80	
Hyperlink	http	`example.com`	80	`/`
rfc3986	http	`example.com`	80	

Table 3: How the accepting targeted parsers handle a strange port number.

`http://example.com:1\u06F0`

Parser	Scheme	Host	Port	Path
CPython	http	`example.com`	10	
Hyperlink	http	`example.com`	10	`/`
furl	http	`example.com`	10	
rfc3986	http	`example.com`	10	

Table 4: How the accepting targeted parsers handle Unicode digits.

Table 3 shows the result of parsing "`http://example.com: +8_0`" with our selected parsers, excluding those that reject.

This peculiar behavior occurs because each of these parsers determines whether a port is valid by attempting to parse it with Python's built-in `int` constructor. However, `int` accepts much more than numeric ASCII strings. For instance, `int` strips all whitespace from either side of its input. This makes URL port numbers a prime location for newline injection. The `int` constructor also removes underscores from between digits, and allows a plus or minus sign to precede the first digit of its input. Further, `int` accepts some Unicode digits, such as the Unicode characters with code points `0x06F0` and `0x0660`, which are both visually similar to a period. Table 4 shows the result of parsing "`http://example.com:1\u06F0`" with our selected parsers, excluding those that reject.

In addition to those parsers blindly using `int` for port parsing, furl is also fooled by this input. In reality, furl also uses `int` to parse ports, but it checks the port string for validity using Python's built-in `str.isdigit` method. This method is Unicode-capable, so Unicode digits can still slip into URL port numbers parsed with furl.

`http://e.vil\@go.od`

Parser	Userinfo	Host	Path
CPython	`e.vil\`	`go.od`	
yarl	`e.vil\`	`go.od`	`/`
Hyperlink	`e.vil\`	`go.od`	`/`
furl	`e.vil\`	`go.od`	
rfc3986		`e.vil`	`%5C@go.od`
urllib3		`e.vil`	`/%5C@go.od`

Table 5: How the targeted parsers handle backslashes in userinfo fields.

4.3 Host Parsing Bugs

The RFC defines a host to be either an IP address or a "reg-name," which is defined to be a string of zero or more of a given set of characters. For example, both the WHATWG standard and the RFC forbid the pipe character (`|`) from appearing in a host. However, all six URL parsers that we tested accept "`http://|.com`" as having host "`|.com`".

4.4 Path Parsing Bugs

The primary difference between the two standards' definitions of URL paths is that the WHATWG standard permits both backslashes and forward slashes as path separator characters, whereas the RFC permits only forward slashes. This difference causes a notable differential (shown in Table 5), which was well-known [Cha; Mos+] before it was rediscovered by dippy_gram.

This differential clearly has security implications, and has at least one CVE assigned to it, CVE-2021-32786. Despite both standards prohibiting the use of backslashes in the userinfo URL component, many URL parsers don't enforce these rules, similar to the lack of character set enforcement demonstrated in Section 4.3. Also noticeable in the table is the automatic percent-encoding employed by rfc3986 and urllib3. While the WHATWG standard clearly identifies when a character is to be percent encoded, the RFC leaves that decision to each parser. In addition to percent-encoding woes, normalization differences involving capitalization, Unicode, and the resolution of path components are also widespread.

5 Next Steps

Our current, preliminary results are promising, but there is still significant progress to be made before this work can be considered complete. We have instrumented and run preliminary experiments with additional parsers (Boost::URL, libcurl, libwget), but work is ongoing to analyze the fuzzing output. Because of differences in permissiveness between these parsers and those written in Python, dippy_gram's signal to noise ratio is lower than is desirable. Due to the abundance of output from dippy_gram, it is difficult to benchmark its output against other fuzzers and program analysis tools. Work is ongoing to determine when two inputs that each cause parser misbehavior are indicative of the same underlying bug(s). Until this issue is solved to a sufficient degree of accuracy, dippy_gram's performance cannot be easily or accurately measured.

Because dippy_gram is written in Python, and interacts with its targets through `afl-showmap`, it is less susceptible to bit rot than a lower-level tool that integrates natively with the execution tracer. Notably, NEZHA is unmaintained and no longer builds with modern libraries. On the other hand, our approach comes with significant performance tradeoffs. Next steps may include rewriting dippy_gram in Rust to integrate with LibAFL [Fio+22].

Once these issues are sorted out, we plan to apply the techniques describe din this paper to more complex protocols. We have already applied this fuzzer to discover inconsistencies in the code generator backends of Apache Daffodil. Work is underway to apply these ideas to search for request smuggling vulnerabilities in HTTP parsers.

6 Conclusion

Due to conflicting standards and a culture of permissiveness, differentials are widespread among URL parsers. Our work demonstrates that grammar-aware, coverage-guided differential fuzzing is well-suited to finding semantic bugs in these programs. This work was made possible by funding from the DARPA GAPS project. URL-specific code is available at `https://github.com/kenballus/url_differential_fuzzing`, and domain-independent

fuzzing code is available at https://github.com/kenballus/diff_fuzz .

References

[Pet+17a] Theofilos Petsios et al. "NEZHA: Efficient Domain-Independent Differential Testing". In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017, pp. 615–632. DOI: 10.1109/SP.2017.27.

[Pet+17b] Theofilos Petsios et al. "SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2155–2168. ISBN: 9781450349468. DOI: 10.1145/3133956.3134073. URL: https://doi.org/10.1145/3133956.3134073.

[Nol+20] Yannic Noller et al. "HyDiff: Hybrid Differential Software Analysis". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 1273–1285. ISBN: 9781450371216. DOI: 10.1145/3377811.3380363. URL: https://doi.org/10.1145/3377811.3380363.

[Jab+21] Bahruz Jabiyev et al. "T-Reqs: HTTP Request Smuggling with Differential Fuzzing". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 1805–1820. ISBN: 9781450384544. DOI: 10.1145/3460120.3485384. URL: https://doi.org/10.1145/3460120.3485384.

[Fio+22] Andrea Fioraldi et al. "LibAFL: A Framework to Build Modular and Reusable Fuzzers". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS '22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 1051–1065. ISBN: 9781450394505. DOI: 10.1145/3548606.3560602. URL: https://doi.org/10.1145/3548606.3560602.

[BFM] Tim Berners-Lee, Roy T. Fielding, and Larry M Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. URL: https://datatracker.ietf.org/doc/rfc3986/.

[Cha] Thomas Chauchefoin. *Security Implications of URL Parsing Differentials*. URL: https://www.sonarsource.com/blog/security-implications-of-url-parsing-differentials/.

[DS] Martin J. Dürst and Michel Suignard. *Internationalized Resource Identifiers (IRIs)*. URL: https://datatracker.ietf.org/doc/rfc3987/.

[Mos+] Noam Moshe et al. *EXPLOITING URL PARSERS: THE GOOD, BAD, AND INCONSISTENT*. URL: https://claroty.com/wp-content/uploads/2022/01/Exploiting-URL-Parsing-Confusion.pdf.

[PyP] PyPI. *Google BigQuery Public Data: PyPI Downloads*. URL: https://bigquery.cloud.google.com/table/bigquery-public-data:pypi.downloads.

[WHA] WHATWG. *URL Living Standard*. URL: https://url.spec.whatwg.org/.

[WTA] Free Wortley, Chris Thompson, and Forrest Allison. *Log4Shell: RCE 0-day exploit found in log4j, a popular Java logging package*. URL: https://www.lunasec.io/docs/blog/log4j-zero-day/.