You Talking To Me?

You Talking To Me? - Li JianTao, starlabs.sg.

- Published: 2021-04-12
- Original: <https://starlabs.sg/blog/2021/04/you-talking-to-me/>
- Preserved from: https://starlabs.sg/blog/2021/04/you-talking-to-me/ (stored) on 2026-08-11
- Capture timestamp: 20220610183920
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

STAR Labs | Blog | You Talking To Me?

[image: image]

Blog

You Talking To Me?

**Apr 12, 2021 **Li JianTao (@cursered)

What is WebDriver and How does it work?

WebDriver is a protocol used for web browser automation. It can drive a browser to perform various tests on web pages as if a real user was navigating through them. It allows simulating user actions such as clicking links, entering text and submitting forms, which can help test if your website is working as intended. It is usually used for front-end testing and web crawling in a headless environment. WebDriver clients (such as Selenium WebDriver) interact with WebDriver servers (e.g. chromedriver, geckodriver) to launch and control browsers. In Capture-the-Flag (CTF) competitions, WebDriver clients are often used to play the role of a victim user (aka. XSS bot) and simulate user interactions to trigger player-supplied XSS payload.

[image: img]

A python script using Selenium Webdriver to launch Chrome and execute JavaScript on example.com

Let me take this example code to explain how WebDriver works. In line 4, the Selenium WebDriver client communicates with chromedriver to launch a Chrome instance, instructs Chrome to visit example.com, executes a piece of JavaScript code, and then quit the browser to end it all.

[image: img]

Sequence diagram of the interactions between WebDriver, chromedriver and Chrome

In this process, WebDriver passes commands to the browser through the driver and receives information via the same route. The driver is responsible for controlling the actual browser, using the browser's built-in support for automation. For example, chromedriver launches a Chrome instance with the option "–remote-debugging-port=0". This causes the Chrome instance to enable the remote debugging feature on a random port for chromedriver to take control of it.

Since the browser vendors themselves create most drivers, the protocol used between driver and browser may vary. Chromium-based browsers use Chrome DevTools Protocol, a set of HTTP and WebSocket endpoints listening on port 9222 by default, whereas Firefox uses its own Marionette Protocol to send and receive JSON-encoded data over a TCP socket. It listens on port 2828 unless otherwise specified.

These drivers are required to follow the W3C standards for WebDriver Protocol and provide consistent REST APIs.

[image: img]

The image shows the default ports the drivers/browsers will be listening to if launched manually, whereas, with WebDriver, the ports are randomized to avoid collisions.

In conclusion, when I use WebDriver to enable the browser visit to some web pages, typically, there will be two open ports on localhost, and at least one of them is an HTTP service providing REST APIs. (Safari is an exception, as its driver and the browser itself are highly integrated within macOS, they talk to each other over XPC Services)

For better readability, "WebDriver" in the following texts refers to the WebDriver Server, in other words, the browser-specific driver(e.g. chromedriver, geckodriver).

Chromedriver from Google

With prior knowledge, I decided to mess around with these WebDrivers to see how far I can go from a security perspective. I started with a very simple sketch script:

[image: img]

There are only two conditions in the script:

- The WebDriver-initiated browser visits our web page
- The browser stays on the page long enough

After two weeks of struggle, I managed to get arbitrary file read and RCE working on Chrome (or more accurately, Chromium-based browsers, including MS Edge and Opera) and Firefox, tested on both Windows and Linux.

Starting from Chrome, I first checked if the randomized ports are accessible from the automated Chrome instance. The answer is yes. The chromedriver REST APIs and Chrome DevTools Protocol(CDP) Server are exposed to the Chrome instance itself.

[image: img]

Potential arbitrary file read via Chrome DevTools Protocol

According to the CDP documentation, the `/json/list` endpoint returns a list of debugging information. If I can somehow read the value of **`webSocketDebuggerUrl`** from the list, I will be able to do everything that CDP is capable of. For example, I can use `Page.navigate` to visit any URL, even in the `file://` scheme, and then use `Runtime.evaluate` to execute arbitrary JavaScript. Combining these two, an attacker can enumerate local directory listings and exfiltrate the contents of any file to a remote server.

[image: img]

But how do we read the **`webSocketDebuggerUrl`** from `http://127.0.0.1:<CDP Port>/json/list`? The response header is quite concise, it only contains `Content-Length` and `Content-Type: application/json; charset=UTF-8`. Speaking of JSON, if the endpoint implemented the JSONP callback feature, we can load it in a script tag from any web page and retrieve data via the callback function. I quickly went through some common param names like `callback`, `cb` and `_callback`, but I have no luck. After diving into the source code, I can confirm that the callback method is not implemented here.

What about DNS rebinding? If the CDP Server does not check for the `Host` header, we can use the DNS rebinding technique to access all CDP endpoints. I tried to change the host to domain `127.0.0.1.xip.io`, which resolves to 127.0.0.1, while the server responded with "Host header is specified and is not an IP address or localhost". By checking the corresponding source code, it is confirmed that the server performs a check for the `Host` header on every request, and I don't see it being bypassed for DNS rebinding.

RCE in chromedriver REST APIs?

Since there is not much I can do with CDP, I moved forward to chromedriver's REST APIs. While reading its documentation and source code, I have found some interesting endpoints that might fit in an exploit chain:

- **GET /session/{sessionid}/source** This endpoint is documented in W3C standards for WebDriver. It returns the source code of the currently active document.
- **GET /sessions** This is a non-W3C standard command implemented by chromedriver unilaterally. It returns every session started by the current chromedriver process. We can find all `{sessionid}` here.
- **POST /session** This one is a W3C standard command for creating a new session. By providing a **`goog:chromeOptions`** object, we can specify the Chrome binary path and even the arguments for chromedriver to start a new Chrome instance with.

The third one seems tempting. With the help of **`strace`**, it didn't take too long to figure out how to execute arbitrary commands by a POST request. From the image below, you can see, our **`-c<python codes>`** argument is parsed and executed perfectly. Some other Chrome arguments are appended by chromedriver, but they are ignored by `Chrome binary` python.

[image: img]

Wow, what an easy RCE!. All it takes to exploit is to scan the port of chromedriver and then send a POST request by form or JS fetch API! But I soon realized it is not as easy as I thought. POST requests issued by browsers will always have an `Origin` header indicating where the request is sent from. Chromedriver has a security check for the `Host` and the `Origin` header.

[image: img]

The check function RequestIsSafeToServe works in this way:

- If chromedriver is launched without option `--allowed-ips`:
- For all requests, the Host header should pass `net::IsLocalhost` check
- If the Origin header is present, its hostname part should pass `net::IsLocalhost` check
- If chromedriver is launched with option `--allowed-ips=<any_ips>`:
- For GET requests, there is no check for the Host header
- For POST requests:
- If the Origin header is not present, there is no check for Host; hence it is impossible to send POST requests without Origin header from browsers.
- If the Origin header is in the format of `IP:port`, the IP must be a local IP or in the `allowed_ips` list. No check for the Host header in this case. It is impossible to send an Origin header from a browser with no `scheme://` in it.
- The Host header and the hostname part of the Origin header should pass `net::IsLocalhost` check

DNS Rebinding to complete the exploit chain

Among all the requests we can send from a browser, it is possible to bypass `RequestIsSafeToServe` with the DNS rebinding attack if the chromedriver is launched with `--allowed-ips` option. That means we have access to every chromedriver REST API that accepts GET requests, including `GET /sessions` and `GET /session/{sessionid}/source`. By combining these three, I can now read the content of CDP's `/json/list`.

[image: img]

The image shows the entire process for an attacker to read `webSocketDebuggerUrl`. The port 9515 and 9222 are only for demonstration purposes. The actual ports are randomized and can be probed by JavaScript. With `webSocketDebuggerUrl`, we can not only read an arbitrary file but also navigate to `http://127.0.0.1:<open port>/` and send POST requests to trigger the RCE since the Host and Origin headers will be legit from `RequestIsSafeToServe` 's perspective.

PoC Videos

[[image: ChromeDriver RCE on Linux]](https://www.youtube.com/watch?v=6y1i6C-RAsE)

[[image: ChromeDriver RCE on Windows]](https://www.youtube.com/watch?v=yl7tbfbIGYA)

Geckodriver from Mozilla

After chromedriver, I started to looked into other WebDrivers to find similar vulnerabilities. Gecko driver is Mozilla's WebDriver for Firefox. Unlike Chrome DevTools Protocol, the protocol it uses to communicate with Firefox is poorly documented. The protocol is called **`Marionette`**, and it is just JSON encoded text in TCP data.

[image: img]

There is no way to send this kind of TCP packet from Firefox. It is a web browser, not pwntools. I also tried to see if Marionette will ignore unrecognized messages like Redis does, so I can smuggle the actual payload in an HTTP request that Firefox can send, but it did not work that way.

Reinforced REST APIs

I took some time to test on geckodriver's REST APIs. Unfortunately, geckodriver can only start one session at a time, so we won't be able to start a new session from our web page, let alone tampering with the path of Firefox binary to execute commands. Although geckodriver does not check the Host header, it implements a stricter check on the Origin and Content-Type headers.

[image: img]

The Origin header must be a local address, and the Content-Type can't be CORS-safelisted. This measure blocks POST requests sent from DNS Rebinding attacks. As for GET requests, no endpoint would return sessionid (GET /sessions in chromedriver is not a W3C standard command), so there is nothing we can do with DNS rebinding.

Splitting the body

So far, both geckodriver and **Marionette** seem to be unexploitable. Just when I was about to give up, something unexpected happened. I was trying to smuggle Marionette commands in an HTTP request; when I repeated the payload string 100,000 times, the geckodriver logged two connections to **Marionette**.

```
<body>
 <form action="#" method="post" enctype="text/plain">
      <textarea name="aaaaaa0" value=""></textarea>
 </form>
  <script>
let params = new URL(location.href).searchParams,

port = 1 * params.get('port')

            document.forms[0].action = `http://127.0.0.1:${port}/`

            document.forms[0].aaaaaa0.value = '54:[0,1,"WebDriver:NewSession",{"browserName":"firefox"}]55:[0,2,"W

            document.forms[0].submit()
 </script>
</body>
```

[image: img]

The first connection was expected; it was the POST request we made. Since it can't be parsed into Marionette's command format `length:[type, message ID, command, parameters]`, an error was thrown.

But where did the second connection come from? How did the packet start in the middle of our repeated payload string from nowhere? I immediately opened WireShark to see what on earth was going on. It turned out that Firefox made two TCP connections for our POST request. The first connection only contained 32KB of the HTTP request body, and the second one sent the remaining part without any HTTP header!

[image: img]

At first, I thought it was some well-known knowledge that I missed out on in that browsers would split large HTTP request body into separate TCP connections. I was wrong. After some testing, only Firefox has such behaviour. I came to realize how much potential this bug has. It allows an attacker to send arbitrary TCP packets from the victim's web browser just by visiting a malicious web page.

[image: img]

The 32KB offset is easy to set with the form type "text/plain" for sending text data. It worked like a charm when we tested against a Redis server. Redis server discarded the first packet because it started with the string" POST", and Redis has protection for such requests. Redis accepted the payload in the second packet. If we want to send binary data, "multipart/form-data" is the choice. Although the randomly generated "boundary" string might become a variable in calculating offset, it can still be brute-forced in just a few tries.

Easy RCE

With the ability to send **Marionette** commands, the same technique we use in chromedriver to read files also works in Firefox. What about RCE? Searching **firefox RCE** on Google brought me to t his article. I soon learned that Firefox has a built-in subprocess JS module available in "chrome-privileged document". All I need to do is navigate to a "chrome://" document and execute one line of JavaScript code.

[image: img]

PoC Videos

[[image: geckodriver RCE on Linux (Firefox 86.0.1)]](https://www.youtube.com/watch?v=ZzGqr5LOJhk)

[[image: geckodriver RCE on Windows (Firefox 86.0.1)]](https://www.youtube.com/watch?v=lMZZOQ0HV08)

Unfortunately, the TCP connection splitting vulnerability has been fixed in Firefox 87.0 as CVE-2021-23982. Samy Kamkar discovered this vulnerability earlier in his research "NAT Slipstreaming".

Other WebDrivers

Since MS Edge and Opera are based on Chromium, their drivers are both derived from chromedriver. With slight modifications, all our payloads for chromedriver also work on both of them. In terms of safaridriver, we didn't find it vulnerable to similar attacks because of its strict check on Host and Origin header.

Takeaway

It has been 14 years since the DNS rebinding attack was discovered and made known to the public. Today, this technique still holds a seat in vulnerability exploit chains from time to time. Generally, HTTP services that only listen to local address are more likely vulnerable to DNS rebinding attacks. We call on developers to validate Host and Origin headers before processing incoming requests. A proper validation can prevent local HTTP services from being exploited by visiting a malicious website.

Timeline

23/03/2021 Firefox 87.0 released, eliminating the TCP connection splitting bug

05/04/2021 Reported **ChromeDriver Privilege Escalation** to Google

08/04/2021 Report marked duplicated with an unresolved issue #3389

12/04/2021 Blog post published

References

https://labs.detectify.com/2017/10/06/guest-blog-dont-leave-your-grid-wide-open/

https://bluec0re.blogspot.com/2018/03/cve-2018-7160-pwning-nodejs-developers.html

https://bugs.chromium.org/p/project-zero/issues/detail?id=1471