# Why Selenium and Why Not?

## Advantages

It’s simply too famous in the world of web automation.

Selenium is great for dynmaic web pages.

## Disadvantages

Selenium has some disadvantages, that sometimes, make it the worst solution:

* **Speed**: Selenium does everything that your browser does, even in headless mode it executes all the JS code. Nowadays, when almost every site uses activity trackers, JS code for JS code. When you go to a URL address - you make dozens of calls to the network. This is not necessary, and yes, make everything too slow!
* **Stability**: Sometimes you just cannot explain why Selenium stopped working. Something has turned to the same webdriver? Was the port busy? Are you already using a browser with an account that you run in Selenium? Is there a parade of planets today? Can this website detect Selenium and ban you? Etc. And worst, it just die giving you a lot of concerns.
* **Scalability**: You can be very upset with the result by launching Selenium in several threads, or browsing multiple pages at the same time.

# Selenium Detection Preventation

## How Detection Bot Works

Many websites is able to detect Selenium using different methods (and often, combine multiple methods). So, we have to understand these methods and find work-arrounds to bypass them as many as possible.

### Detect Predefined JavaScript Variables

Basically, Selenium detection works by testing for predefined JavaScript variables which appear when running with Selenium.

The bot usually look anything containing word "selenium", "webdriver", etc. in any of the variables (on window object), and also document variables called $cdc\_ and $wdc\_. All of this depends on which browser you are on beacause different browsers expose different things.

Here is pseudocode which demonstrates some of the techniques that bot networks might use:

runBotDetection = function () {

var documentDetectionKeys = [

"\_\_webdriver\_evaluate",

"\_\_selenium\_evaluate",

"\_\_webdriver\_script\_function",

"\_\_webdriver\_script\_func",

"\_\_webdriver\_script\_fn",

"\_\_fxdriver\_evaluate",

"\_\_driver\_unwrapped",

"\_\_webdriver\_unwrapped",

"\_\_driver\_evaluate",

"\_\_selenium\_unwrapped",

"\_\_fxdriver\_unwrapped",

];

var windowDetectionKeys = [

"\_phantom",

"\_\_nightmare",

"\_selenium",

"callPhantom",

"callSelenium",

"\_Selenium\_IDE\_Recorder",

];

for (const windowDetectionKey in windowDetectionKeys) {

const windowDetectionKeyValue = windowDetectionKeys[windowDetectionKey];

if (window[windowDetectionKeyValue]) {

return true;

}

};

for (const documentDetectionKey in documentDetectionKeys) {

const documentDetectionKeyValue = documentDetectionKeys[documentDetectionKey];

if (window['document'][documentDetectionKeyValue]) {

return true;

}

};

for (const documentKey in window['document']) {

if (documentKey.match(/\$[a-z]dc\_/) && window['document'][documentKey]['cache\_']) {

return true;

}

}

if (window['external'] && window['external'].toString() && (window['external'].toString()['indexOf']('Sequentum') != -1)) return true;

if (window['document']['documentElement']['getAttribute']('selenium')) return true;

if (window['document']['documentElement']['getAttribute']('webdriver')) return true;

if (window['document']['documentElement']['getAttribute']('driver')) return true;

return false;

};

Refs:

<https://stackoverflow.com/a/41220267>

### Detect Malicious Useragents

### Detect Webdriver Navigator

### Detect Plugins

navigator.plugins returns an array of plugins present in the browser. Typically, on Chrome we find default plugins, such as Chrome PDF viewer or Google Native Client.

On the opposite, in headless mode of chromedriver, the array returned contains no plugin.

A check for the presence of Plugins can be done through:

if(navigator.plugins.length == 0) {

console.log("It may be Chrome headless");

}

Ref: <https://stackoverflow.com/a/56529616>

### Detect Languages

In Chrome, navigator.language and navigator.languages obtain languages used by the user. The first one is the language of the browser UI, and the second one is an array of string representing the user’s preferred languages.

However, in headless mode of chromedriver, navigator.languages returns an empty string.

A check for the presence of Languages can be done through:

if(navigator.languages == "") {

console.log("Chrome headless detected");

}

Ref: <https://stackoverflow.com/a/56529616>

### Detect WebGL

WebGL is an API to perform 3D rendering in an HTML canvas. With this API, it is possible to query for the vendor of the graphic driver as well as the renderer of the graphic driver. With a vanilla Chrome and Linux, we can obtain the following values for renderer and vendor: Google SwiftShader and Google Inc.. In headless mode, we can obtain Mesa OffScreen, which is the technology used for rendering without using any sort of window system and Brian Paul, which is the program that started the open source Mesa graphics library.

A check for the presence of WebGL can be done through:

var canvas = document.createElement('canvas');

var gl = canvas.getContext('webgl');

var debugInfo = gl.getExtension('WEBGL\_debug\_renderer\_info');

var vendor = gl.getParameter(debugInfo.UNMASKED\_VENDOR\_WEBGL);

var renderer = gl.getParameter(debugInfo.UNMASKED\_RENDERER\_WEBGL);

if(vendor == "Brian Paul" && renderer == "Mesa OffScreen") {

console.log("Chrome headless detected");

}

Not all Chrome headless will have the same values for vendor and renderer. Others keep values that could also be found on non headless version. However, Mesa Offscreen and Brian Paul indicates the presence of the headless version.

Ref: <https://stackoverflow.com/a/56529616>

### Others

## How To Pass Detection Bot

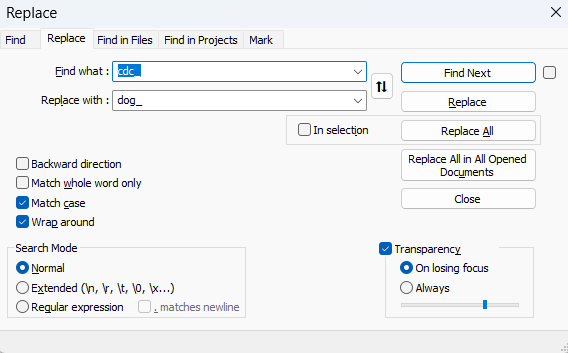
### Pass Detection of Predefined JavaScript Variables

Two ways:

* Replace the cdc\_ string in chromedriver.exe with any other string. For safe, the chosen string should has the same number of characters as the search string (e.g., dog\_).

For how to replace cdc\_ string, check [here](https://stackoverflow.com/a/52108199).

My way: Open chromedriver.exe with Notepadd++ and replace string in an usual way.



* Tried. It works perfectly for Shopee website.
* Download chromedriver source code, modify chromedriver and re-compile $cdc\_ under different name.
* Not tried yet

### Use Valid Useragents

In Python, you can do this using the integrated chromedriver option:

useragent = 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.97 Safari/537.36'})

options.add\_argument('user-agent={}'.format(useragent))

driver = webdriver.Chrome(executable\_path="path-to-chromedriver", options=options)

Or manually via the Chrome Devtools Protocol command:

driver = webdriver.Chrome(executable\_path="path-to-chromedriver")

print(driver.execute\_script("return navigator.userAgent;"))

# Setting useragent as Chrome/83.0.4103.97

driver.execute\_cdp\_cmd('Network.setUserAgentOverride', {"userAgent": 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.97 Safari/537.36'})

print(driver.execute\_script("return navigator.userAgent;"))

# Setting user agent as Chrome/83.0.4103.53

driver.execute\_cdp\_cmd('Network.setUserAgentOverride', {"userAgent": 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.53 Safari/537.36'})

print(driver.execute\_script("return navigator.userAgent;"))

driver.get('https://www.httpbin.org/headers')

Output:

Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.106 Safari/537.36

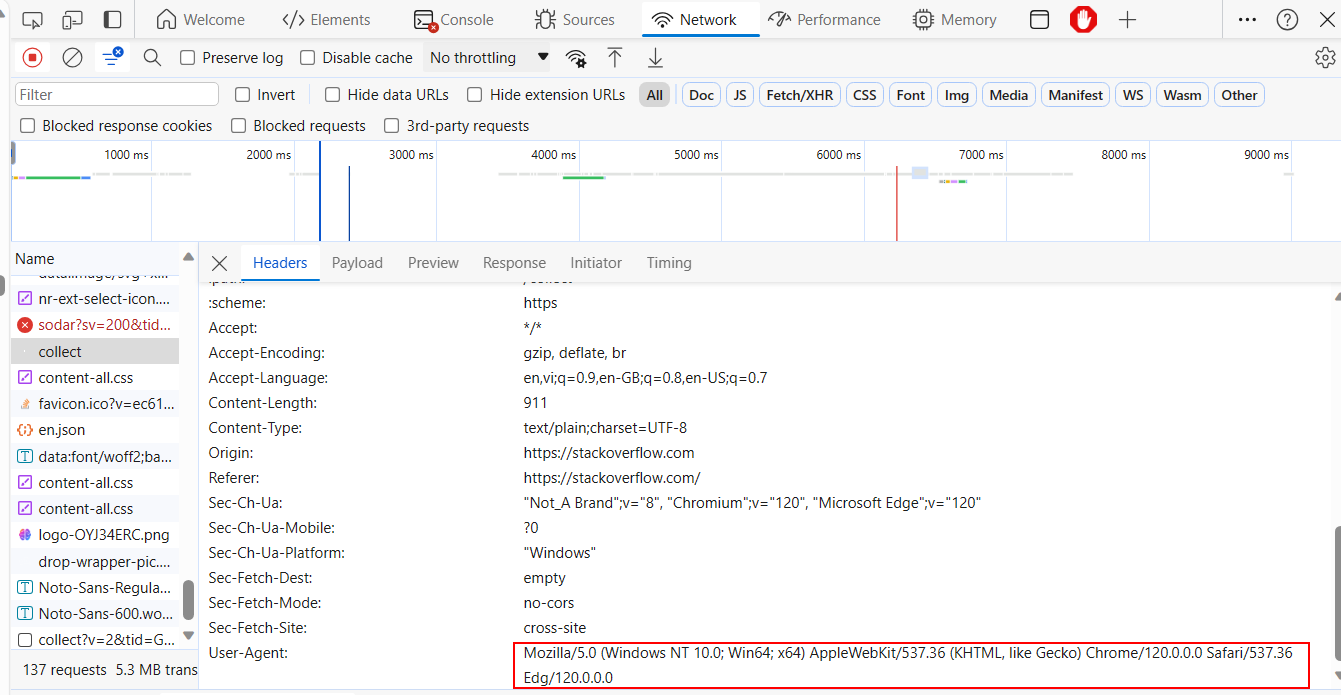
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.97 Safari/537.36

Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.53 Safari/537.36

**Notes**:

* Useragent changes depending on different web browsers and their version. So, make sure you choose a up-to-date one.

You can esily find valid useragent using ChromeDev Tool:



### Fake Plugins

### Fake Langauges

### Fake WebGL

### Disable Webdriver Navigator

In Python, you can do this using the integrated chromedriver option:

options.add\_argument('--disable-blink-features=AutomationControlled')

driver = webdriver.Chrome(executable\_path="path-to-chromedriver", options=options)

Or manually via the Chrome Devtools Protocol command:

driver.execute\_cdp\_cmd("Page.addScriptToEvaluateOnNewDocument", {

"source": """

Object.defineProperty(navigator, 'webdriver', {

get: () => undefined

})

"""

})

Or:

driver.execute\_script("Object.defineProperty(navigator, 'webdriver', {get: () => undefined})")

driver.execute\_cdp\_cmd("Page.addScriptToEvaluateOnNewDocument", {

   "source":

      "const newProto = navigator.\_\_proto\_\_;"

      "delete newProto.webdriver;"

      "navigator.\_\_proto\_\_ = newProto;"

})

### Others

* Exclude the collection of enable-automation switches.
* Turn off useAutomationExtension.

Python code:

self.options.add\_experimental\_option('excludeSwitches', ['enable-automation'])

self.options.add\_experimental\_option('useAutomationExtension', False)

## How To Pass CloudFlare Detection

### Solution 1: Simple

Might not work for some websites.

<https://stackoverflow.com/a/76575381>

### Solution 2: Advanced

* Using [**Undetected-Selenium**](#_Undetected-Chromedriver): Work well on most websites, such as dexscreener.com.

Ref: <https://scrapfly.io/blog/web-scraping-without-blocking-using-undetected-chromedriver/>

* Using [**SeleniumBase**](#_SeleniumBase): Might not work for some websites, such as dexscreener.com.

Ref: <https://stackoverflow.com/a/76575463>

WARNING:

**FlareSolver**: Doesn't work

<https://github.com/FlareSolverr/FlareSolverr>

<https://scrapeops.io/python-web-scraping-playbook/python-flaresolverr/>

# Webdriver Configurations

## Run Selenium in Background

One of the best thing about Selenium compared to many other automation solutions is that it can work in background (a.k.a, headless mode). This means you can run your webdirver while doing other things with your PC.

In Python, to run Chrome webdriver in headless mode:

options.add\_argument("--headless")

driver = webdriver.Chrome(executable\_path="path-to-chromedriver", options=options)

## Start Web Browser with Full Window Size

In Python, to start chrome webdriver in maximum size:

options.add\_argument("start-maximized")

driver = webdriver.Chrome(executable\_path="path-to-chromedrive", options=options)

## Handle Cookies

There are many use cases when you want to reuse cookies from your favorite web browser. For examples, logging in a website with username and password; or checking order history, viewing cart, payments, etc. in an online shopping store.

* In Python, to re-use cookies in Chrome driver:

profile\_path = os.getenv('LOCALAPPDATA') + "\\Google\\Chrome\\User Data"

options.add\_argument("user-data-dir=" + profile\_path")

driver = webdriver.Chrome(executable\_path="path-to-chromedriver", options=options)

Note: The above code is for Chrome. For Firefox, it is much more complicated.

* To add, get and delete cookies, check:
  + APIs at [here](https://www.seleniumeasy.com/selenium-tutorials/how-to-handle-cookies-in-selenium-webdriver).
  + Example code (in Python) at [here](https://github.com/ArturSpirin/YouTube-WebDriver-Tutorials/blob/master/Cookies.py).
  + Another simple way to delete cookies at [here](https://stackoverflow.com/a/55541639).
* To delete caches, check [here](https://stackoverflow.com/a/50456808).

## Keep Chrome Webdriver Opening After The Script Is Completed

Chrome webdriver will be terminated after the scripts run to the end. In many cases, you don’t want that because you want to see the final result of all automation steps.

In Python, to keep Chrome webdriver (and its window) opening:

options.add\_experimental\_option("detach", True)

driver = webdriver.Chrome(executable\_path="path-to-chromedriver", options=options)

Note: Firefox webdriver already keeps opening its process and window, so no need to add anything.

# Webdriver Control

## Perform Keystroke

Controlling a web page can be must easier if keystrokes are used instead of clicking.

There are some ways to perform a keystroke.

* Use driver.find\_element\_by\*\*\*.send\_keys() if you know exactly the element to send the key to
* Use ActionChains class if you don’t have any element to send the key to. Note that this class not only allows keystrokes but also supports mouse movement and various clicking actions.

For example in Python:

# Copy text (clicking on the page body, blackout all text and copy it)

driver.find\_element\_by\_tag\_name('body').send\_keys(Keys.CONTROL + "a" + "c")

# Paste text

from selenium.webdriver.common.action\_chains import ActionChains

ActionChains(driver).key\_down(Keys.CONTROL).send\_keys("v").key\_up(Keys.CONTROL).perform()

# Note: Rememmber to key\_up after each key\_down. Else, later keystrokes will be affected.

## Wait For Elements Loaded

Web page items need time to be fully loaded and displayed.

<http://allselenium.info/wait-for-elements-python-selenium-webdriver/>

<https://stackoverflow.com/a/27112797/14835442>

## Wait For A Page Fully Loaded

We can’t. All we can is to check if a specific element (or some) is loaded, not the whole page.

Ref: <https://stackoverflow.com/questions/26566799/wait-until-page-is-loaded-with-selenium-webdriver-for-python>

**Note:** Following ways do not work:

* Check READY state in JavaScript:

def wait\_page\_loaded():   # NOT WORK. Also cause Selection detection by websites

    ''' Wait until the page is completely loaded

    '''

    WebDriverWait(webdriver, 100).until(

        lambda driver: driver.execute\_script('return document.readyState') == 'complete'

    )

Ref: <https://stackoverflow.com/a/73035261>

## Open in New Tab

It’s useless to use keystroke to simulate "Ctrl + Tab" to open a new tab – either with Chrome or Firefox. (That seem a limitation of Selenium).

In Python, to open a new tab in Chrome or Firefox:

driver.execute\_script("window.open('');") # Open in new tab

driver.switch\_to.window(driver.window\_handles[-1]) # Switch to the last opened tab handle

driver.get('your-url') # Now you can go to the URL you want

## Scroll Down/Up

<https://www.browserstack.com/guide/selenium-scroll-tutorial>

## Debug Issue "unable to locate element" in Headless Mode

<https://stackoverflow.com/a/63049009/14835442>

## Capturing REST API's Requests and Responses

This is extremely useful. We can scrape multiple pieces of data from web pages through REST API response. This is much faster and more stable compared to the traditional Selenium ways – locating each element address (via Xpath, class name, etc.) and get its value – once by once.

|  |  |
| --- | --- |
|  |  |

Unluckily, Selenium doesn't OFFICIALLY support this feature. Its team doesn't want to do this because they want Selenium webdriver focus on automating web-based applications, not on API testing.

Luckily, with some work-arround steps, we can still capture resquests and responses based on our needs.

### Via Network Event Logs

Selenium logs all **network events**. You can find REST API requests and responses in these logs. What you have to do is:

1. Enable network event logs
2. Get all log (particularly, the PERFORMANCE log)
3. Filter the log to get the request/response of the REST API you need. For some parts of the response, e.g. response body, you’ll need to execute commands from Chrome DevTools Protocol.

Ref: <https://stackoverflow.com/a/77065745>

* Tried. It works

### Via Proxy Tool

You can use a proxy tool to capture response body.

For example, BrowserMobProxy can record API requests and responses in .har format that you can later open and view. It will contain all headers, data and timings

* Not tried yet.

### Via [selenium-wire](#_Selenium-Wire) Package

It’s a wrapper of Selenium which add supports for **capturing and** **inspecting REST API’s requests and responses** made by the browser.

Not just inspecting, it also allows to **modify REST API requests and responses** using interceptors.

* Tried. It works.

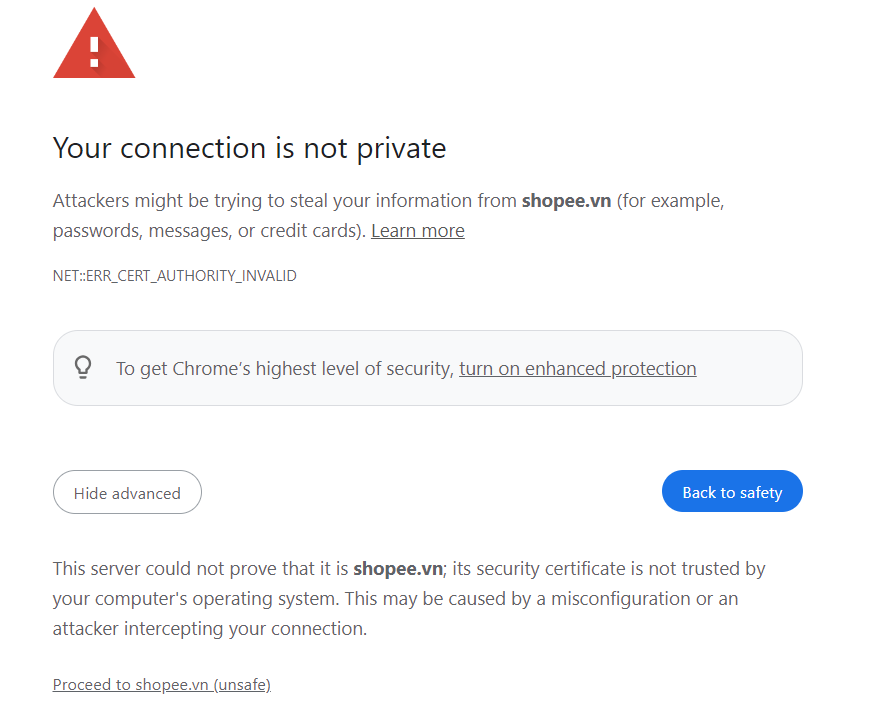
However, since it's a wrapper of the original Selenium, I concern about its side-effects in terms of my project’s coding styles and Selenium’s feature limitations.

For examples:

* Selenium is detected by websites. By contrast, the same code base, but with the original Selenium, is out of detection.

Selenium-wire seems to know that, so it integrates "undetected-chromedriver" lib, particularly in the seleniumwire.undetected\_chromedriver package.

* When using seleniumwire.undetected\_chromedriver package, get error "Your connection is not private" when browsing Shopee website.



Selenium-wire knows that, and explain and solution [here](https://github.com/wkeeling/selenium-wire#certificates). I tried all, but none of them work.

Refs:

<https://www.zenrows.com/blog/selenium-wire#request-response-objects>

<https://scrapeops.io/selenium-web-scraping-playbook/python-selenium-wire/>

## Find Button By Text Name

Instead of using CSS class or Xpath, we can get elements and perform operations on them using their displayed name (or labels).

Coding with this way is simpler and easier to read. But more important, we can get rid of dealing with complex dynamic elements where their CSS class or Xpath can change based on different conditions.

<https://stackoverflow.com/a/49906334>

## Using BeautifulSoup with Selenium

<https://stackoverflow.com/a/62256021>

# Chrome DevTools Protocol (CDP)

<https://applitools.com/blog/selenium-4-chrome-devtools/>

<https://rahulshettyacademy.com/blog/index.php/2021/11/04/selenium-4-key-feature-network-interception/>

# WebDriver BiDirectional Protocol (WebDriver BiDi)

<https://www.selenium.dev/documentation/webdriver/bidirectional/>

<https://www.awesome-testing.com/2023/04/exploring-selenium-bidi-functionality>

# Selenium Wrappers

There are dozens of Selenium wrappers which add more features to the raw Selenium, or help it easier to use.

## SeleniumBase

<https://github.com/seleniumbase/SeleniumBase?tab=readme-ov-file#install_seleniumbase>

It focuses on **automatically handling common WebDriver actions** such as launching web browsers before tests, saving screenshots during failures, closing web browsers after tests, downloading webdrivers as needed, and [more](https://github.com/seleniumbase/SeleniumBase/blob/master/help_docs/features_list.md).

## Selenium-Wire

<https://github.com/wkeeling/selenium-wire>

It focuses on adding supports for **capturing and** **inspecting REST API’s requests and responses** made by the browser.

## Undetected-Chromedriver

<https://github.com/ultrafunkamsterdam/undetected-chromedriver>

It focuses on **bypassing bot detection** – a wide range of bots: CloudFlare, Distill Network, Imperva, DataDome, Botprotect.io.

# Performance Improvement

<https://www.zenrows.com/blog/selenium-slow#reboot-your-servers-programmatically>

<https://www.selenium.dev/documentation/grid/>

# Selenium Alternatives

## Puppeteer

<https://github.com/puppeteer/puppeteer>

# Web Scraping Without Selenium

<https://community.dataquest.io/t/web-scraping-without-selenium/456297>

<https://stackoverflow.com/questions/72340316/can-i-use-post-method-in-requests-lib-on-this-binance-site>