Chrome 0-day exploit CVE-2019-13720 used in Operation WizardOpium

By AMR

Published: 2019-11-01 · Archived: 2026-04-05 12:59:56 UTC

Executive summary

Kaspersky Exploit Prevention is a component part of Kaspersky products that has successfully detected a number of zero-day attacks in the past. Recently, it caught a new unknown exploit for Google's Chrome browser. We promptly reported this to the Google Chrome security team. After reviewing of the PoC we provided, Google confirmed there was a zero-day vulnerability and assigned it CVE-2019-13720. Google has released Chrome version 78.0.3904.87 for Windows, Mac, and Linux and we recommend all Chrome users to update to this latest version as soon as possible! You can read Google's bulletin by clicking here.

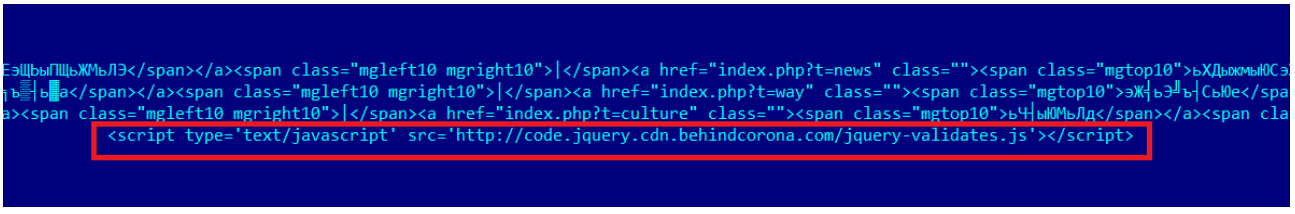
Kaspersky endpoint products detect the exploit with the help of the exploit prevention component. The verdict for this attack is Exploit.Win32.Generic.

We are calling these attacks Operation WizardOpium. So far, we have been unable to establish a definitive link with any known threat actors. There are certain very weak code similarities with Lazarus attacks, although these could very well be a false flag. The profile of the targeted website is more in line with earlier DarkHotel attacks that have recently deployed similar false flag attacks.

More details about CVE-2019-13720 and recent DarkHotel false flag attacks are available to customers of Kaspersky Intelligence Reporting. For more information, contact: intelreports@kaspersky.com.

Technical details

The attack leverages a waterhole-style injection on a Korean-language news portal. A malicious JavaScript code was inserted in the main page, which in turn, loads a profiling script from a remote site.



Redirect to the exploit landing page

The main index page hosted a small JavaScript tag that loaded a remote script from hxxp://code.jquery.cdn.behindcorona[.]com/.

The script then loads another script named *.charlie.XXXXXXXX.js*. This JavaScript checks if the victim's system can be infected by performing a comparison with the browser's user agent, which should run on a *64-bit* version of *Windows* and not be a *WOW64* process; it also tries to get the browser's name and version. The vulnerability tries to exploit the bug in *Google Chrome* browser and the script checks if the version is greater or equal to 65 (current Chrome version is 78):

```
if (navigator.userAgent.indexOf("Win64") == -1 || navigator.userAgent.indexOf("WOW64") != -1)
        return ;

if (navigator.userAgent.indexOf("Windows NT 6.1") == -1)
        return ;

let r = navigator.userAgent.indexOf("Chrome/");

if (r == -1)
        return ;

if (parseInt(navigator.userAgent.substr(r + "Chrome/".length, 3)) < 65)
        return ;
```

Chrome version checks in the profiling script (.charlie.XXXXXXXX.js)

If the browser version checks out, the script starts performing a number of AJAX requests to the attacker's controlled server (*behindcorona[.]com*) where a path name points to the argument that is passed to the script (*xxxxxxx.php*). The first request is necessary to obtain some important information for further use. This information includes several hex-encoded strings that tell the script how many chunks of the actual exploit code should be downloaded from the server, as well as a URL to the image file that embeds a key for the final payload and RC4 key to decrypt these chunks of the exploit's code.

404	HTTP	behindcorona.com	/favicon.ico
200	HTTP	behindcorona.com	[redacted]
200	HTTP	behindcorona.com	
200	HTTP	behindcorona.com	
200	HTTP	behindcorona.com	
200	HTTP	behindcorona.com	
200	HTTP	behindcorona.com	
200	HTTP	behindcorona.com	
200	HTTP	behindcorona.com	
200	HTTP	behindcorona.com	
200	HTTP	behindcorona.com	

Exploitation chain – AJAX requests to xxxxxxx.php

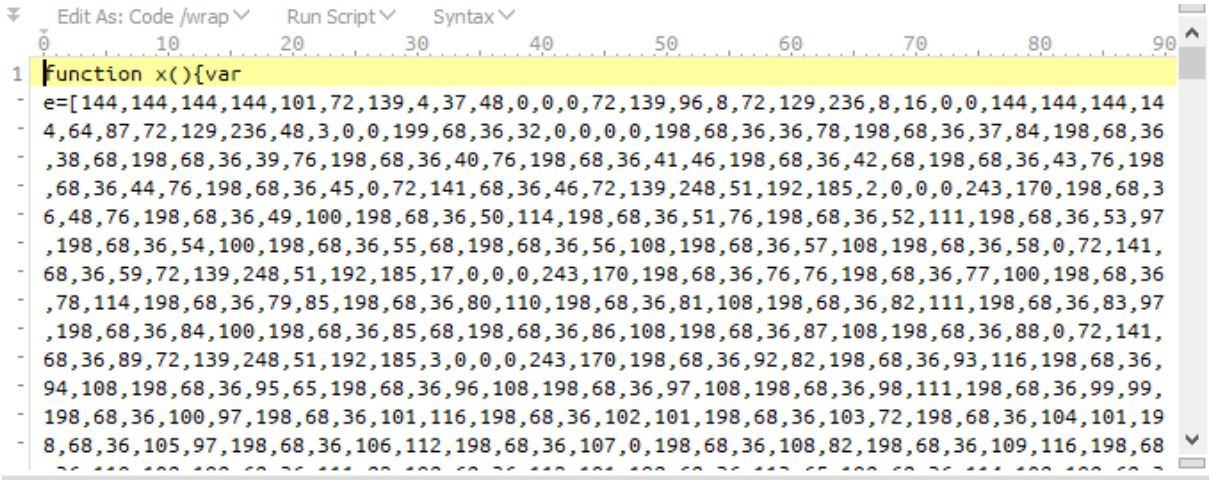
After downloading all the chunks, the *RC4* script decrypts and concatenates all the parts together, which gives the attacker a new JavaScript code containing the full browser exploit. To decrypt the parts, the previously retrieved RC4 key is used.

```
try {
    var n = navigator.userAgent.split("Chrome/")[1].split(" Safari/")[0];
    n = parseInt(n.substr(0, 2));
    if (n != 77 && n != 76) {
        return
    }
```

One more version check

The browser exploit script is obfuscated; after de-obfuscation we observed a few peculiar things:

1. 1 Another check is made against the user agent's string – this time it checks that the browser version is 76 or 77. It could mean that the exploit authors have only worked on these versions (a previous exploitation stage checked for version number 65 or newer) or that other exploits have been used in the past for older Chrome versions.



Obfuscated exploit code

2. 2 There are a few functions that operate on the browser's built-in *BigInt* class, which is useful for doing 64-bit arithmetic inside JavaScript code, for example, to work with native pointers in a 64-bit environment. Usually, exploit developers implements their own functions for doing this by working with 32-bit numbers. However, in this case, *BigInt* is used, which should be faster because it's implemented natively in the browser's code. The exploit developers don't use all 64 bits here, but instead operate on a smaller range of numbers. This is why they implement a few functions to work with higher/lower parts of the number.

```
function O(e) {
    var t = D(e);
    var n = (BigInt(1) << BigInt(21)) - BigInt(1);
    var r = (e & n) >> BigInt(14);
    var i = t + (BigInt(4096) + r * BigInt(32));
    return i
}

function k(e, t) {
    var n = D(e);
    var r = t << BigInt(14);
    return n + r
}

function E(e) {
    var t = D(e);
    var n = (BigInt(1) << BigInt(21)) - BigInt(1);
    var r = (e & n) >> BigInt(14);
    return r
}
```

Snippet of code to work with 64-bit numbers

3. 3 There are many functions and variables that are not used in the actual code. This usually means that they were used for debugging code and were then left behind when the code was moved to production.
4. 4 The majority of the code uses several classes related to a certain vulnerable component of the browser. As this bug has still not been fixed, we are not including details about the specific vulnerable component here.
5. 5 There are a few big arrays with numbers that represent a shellcode block and an embedded PE image.

The analysis we have provided here is deliberately brief due to vulnerability disclosure principles. The exploit used a *race condition* bug between two threads due to *missing proper synchronization* between them. It gives an attacker an a *Use-After-Free (UaF)* condition that is very dangerous because it can lead to code execution scenarios, which is exactly what happens in our case.

The exploit first tries to trigger *UaF* to perform an information leak about important 64-bit addresses (as a pointer). This results in a few things: 1) if an address is leaked successfully, it means the exploit is working correctly; 2) a leaked address is used to know where the heap/stack is located and that defeats the *address space layout randomization (ASLR)* technique; 3) a few other useful pointers for further exploitation could be located by searching near this address.

After that it tries to create a bunch of large objects using a recursive function. This is done to make some deterministic heap layout, which is important for a successful exploitation. At the same time, it attempts to utilize a heap spraying technique that aims to reuse the same pointer that was freed earlier in the UaF part. This trick could be used to cause confusion and give the attacker the ability to operate on two different objects (from a JavaScript code perspective), though in reality they are located in the same memory region.

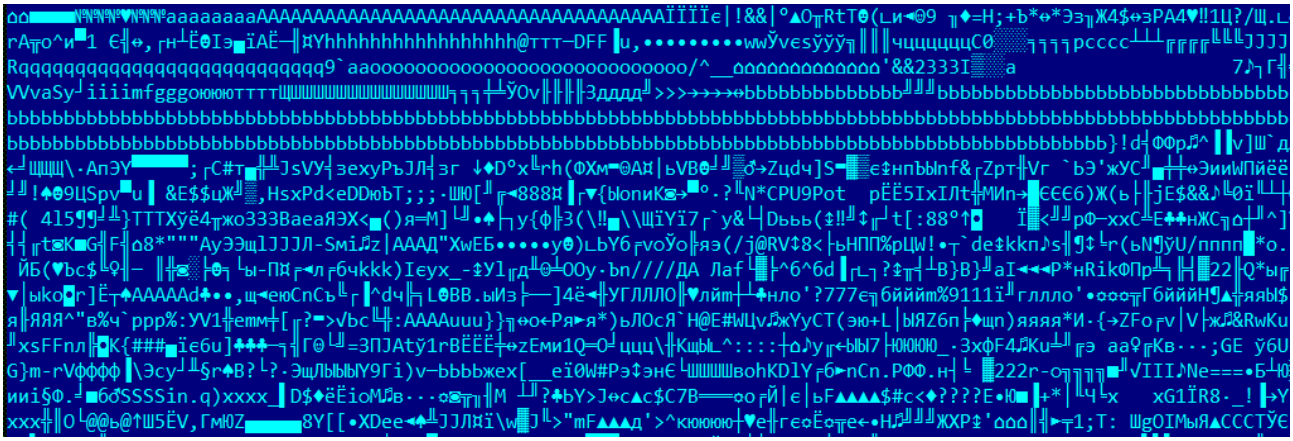
The exploit attempts to perform numerous operations to allocate/free memory along with other techniques that eventually give the attackers an arbitrary read/write primitive. This is used to craft a special object that can be used with *WebAssembly* and *FileReader* together to perform code execution for the embedded shellcode payload.

```
00000000: 90909090              nop
00000004: 6548                  dec         eax
00000006: 8B042530000000        mov         eax,[000000030]
0000000D: 48                    dec         eax
0000000E: 8B6008                mov         esp,[eax][8]
00000011: 48                    dec         eax
00000012: 81EC08100000          sub         esp,000001008
00000018: 90909090              nop
0000001C: 40                    inc         eax
0000001D: 57                    push        edi
0000001E: 48                    dec         eax
0000001F: 81EC30030000          sub         esp,000000330 ;'  ♥0'
00000025: C744242000000000      mov         d,[esp][020],0
0000002D: C64424244E            mov         b,[esp][024],04E ;'N'
00000032: C644242554            mov         b,[esp][025],054 ;'T'
00000037: C644242644            mov         b,[esp][026],044 ;'D'
0000003C: C64424274C            mov         b,[esp][027],04C ;'L'
00000041: C64424284C            mov         b,[esp][028],04C ;'L'
00000046: C64424292E            mov         b,[esp][029],02E ;'.'
0000004B: C644242A44            mov         b,[esp][02A],044 ;'D'
00000050: C644242B4C            mov         b,[esp][02B],04C ;'L'
00000055: C644242C4C            mov         b,[esp][02C],04C ;'L'
0000005A: C644242D00            mov         b,[esp][02D],0
0000005F: 48                    dec         eax
```

First stage shellcode

Payload description

The final payload is downloaded as an encrypted binary (worst.jpg) that is decrypted by the shellcode.



Encrypted payload – worst.jpg

After decryption, the malware module is dropped as updata.exe to disk and executed. For persistence the malware installs tasks in Windows Task Scheduler.

The payload ‘installer’ is a RAR SFX archive, with the following information:

File size: 293,403
MD5: 8f3cd9299b2f241daf1f5057ba0b9054

SHA256: 35373d07c2e408838812ff210aa28d90e97e38f2d0132a86085b0d54256cc1cd

The archive contains two files:

```
;The comment below contains SFX script commands
        Path=%programdata%
        Setup=iohelper.exe
        Silent=1
        Overwrite=1
        Testing         msdisp64.exe                    OK
        Testing         iohelper.exe                    OK
        All OK
```

File name: iohelper.exe
MD5: 27e941683d09a7405a9e806cc7d156c9
SHA256: 8fb2558765cf648305493e1dfea7a2b26f4fc8f44ff72c95e9165a904a9a6a48

File name: msdisp64.exe
MD5: f614909fbd57ece81d00b01958338ec2
SHA256: cafe8f704095b1f5e0a885f75b1b41a7395a1c62fd893ef44348f9702b3a0deb

Both files were compiled at the same time, which if we are to believe the timestamp, was “Tue Oct 8 01:49:31 2019”.
The main module (msdisp64.exe) tries to download the next stage from a hardcoded C2 server set. The next stages are located on the C2 server in folders with the victim computer names, so the threat actors have information about which machines were infected and place the next stage modules in specific folders on the C2 server.

More details about this attack are available to customers of Kaspersky Intelligence Reporting. For more information, contact: intelreports@kaspersky.com.

IoCs

- behindcorona[.]com
- code.jquery.cdn.behindcorona[.]com
- 8f3cd9299b2f241daf1f5057ba0b9054
- 35373d07c2e408838812ff210aa28d90e97e38f2d0132a86085b0d54256cc1cd
- 27e941683d09a7405a9e806cc7d156c9
- 8fb2558765cf648305493e1dfea7a2b26f4fc8f44ff72c95e9165a904a9a6a48
- f614909fbd57ece81d00b01958338ec2
- cafe8f704095b1f5e0a885f75b1b41a7395a1c62fd893ef44348f9702b3a0deb
- kennethosborne@protonmail.com
