POSTED BY
EXODUS INTEL VRT

POSTED ON
MARCH 20, 2019

POSTED UNDER
UNCATEGORIZED

CVE-2019-5786: ANALYSIS & EXPLOITATION OF THE RECENTLY PATCHED CHROME VULNERABILITY

This post provides detailed analysis and an exploit achieving remote code execution for the recently fixed Chrome vulnerability that was observed by Google to be exploited in the wild.

Search ...

RECENT POSTS

Patch-gapping Google Chrome

Pwn2Own 2019: Microsoft Edge Sandbox Escape (CVE-2019-0938). Part 2

Pwn2Own 2019: Microsoft Edge Renderer Exploitation

Author: István Kurucsai

Patch Analysis

The release notes from Google are short on information as usual:

> [$N/A][936448] High CVE-2019-5786: Use-after-free in FileReader. Reported by Clement Lecigne of Google's Threat Analysis Group on 2019-02-27

As described on MDN, the "FileReader object lets web applications asynchronously read the contents of files (or raw data buffers) stored on the user's computer, using File or Blob objects to specify the file or data to read". It can be used to read the contents of files selected in a file open dialog by the user or Blobs created by script code. An example usage is shown in below.

```
let reader = new FileReader();

reader.onloadend = function(evt) {
  console.log(`contents as an ArrayBuffer: ${evt.target.
}

reader.onprogress = function(evt) {
  console.log(`read ${evt.target.result.byteLength} byte
}

let contents = "filecontents";
f = new File([contents], "a.txt");
reader.readAsArrayBuffer(f);
```

(CVE-2019-0940). Part 1

Windows Within Windows – Escaping The Chrome Sandbox With a Win32k NDay

A window of opportunity: exploiting a Chrome 1day vulnerability

RECENT COMMENTS

Stagefright Patch Incomplete Leaving Android Devices Still Exposed | The Root Shell on

It is important to note that the File or Blob contents are read asynchronously and the user JS code is notified of the progress via callbacks. The *onprogress* event may be fired multiple times while the reading is in progress, giving access to the contents read so far. The *onloadend* event is triggered once the operation is completed, either in success or failure.

Searching for the issue number in the Chromium git logs quickly reveals the patch for the vulnerability, which alters a single function. The original, vulnerable version is shown below.

```
1  DOMArrayBuffer* FileReaderLoader::ArrayBufferResult() {
2    DCHECK_EQ(read_type_, kReadAsArrayBuffer);
3    if (array_buffer_result_)
4      return array_buffer_result_;
5
6    // If the loading is not started or an error occurs, r
7    if (!raw_data_ || error_code_ != FileErrorCode::kOK)
8      return nullptr;
9
10   DOMArrayBuffer* result = DOMArrayBuffer::Create(raw_da
11   if (finished_loading_) {
12     array_buffer_result_ = result;
13     AdjustReportedMemoryUsageToV8(
14         -1 * static_cast<int64_t>(raw_data_->ByteLength(
15     raw_data_.reset();
16   }
17   return result;
18 }
```

This function gets called each time the result property is accessed in a callback after a *FileReader.readAsArrayBuffer* call in JavaScript.

Stagefright: Mission Accomplished?

Stagefright Patch Incomplete Leaving Android Devices Still Exposed | Threatpost | The first stop for security news on Stagefright: Mission Accomplished?

Tails live OS affected by critical zero-day vulnerabilities on Silver Bullets and Fairy Tails

Are Tor and Tails Safe? | SxiSpiGrl on Silver Bullets and Fairy Tails

While the object hierarchy around the C++ implementation of ArrayBuffers is relatively complicated, the important pieces are described below. Note that the C++ namespaces of the different classes are included so that distinguishing between objects implemented in Chromium (the WTF and blink namespaces) and v8 (everything under the v8 namespace) is easier.

- **WTF::ArrayBuffer**: the embedder-side (Chromium) implementation of the ArrayBuffer concept. **WTF::ArrayBuffer** objects are reference counted and contain the raw pointer to their underlying memory buffer, which is freed when the reference count of an ArrayBuffer reaches 0.
- **blink::DOMArrayBufferBase**: a garbage collected class containing a smart pointer to a **WTF::ArrayBuffer**.
- **blink::DOMArrayBuffer**: class inheriting from **blink::DOMArrayBufferBase**, describing an ArrayBuffer in Chromium. Represented in the JavaScript engine by a **v8::internal::JSArrayBuffer** instance.
- **WTF::ArrayBufferBuilder**: helper class to construct a **WTF::ArrayBuffer** incrementally. Holds a smart pointer to the ArrayBuffer.
- **blink::FileReaderLoader**: responsible for loading the File or Blob contents. Uses **WTF::ArrayBufferBuilder** to build

Scott Herbert (@Scott_Herbert) on Silver Bullets and Fairy Tails

ARCHIVES

September 2019
May 2019
April 2019
March 2019
January 2019
October 2018
September 2018
October 2017
July 2017

the ArrayBuffer as the data is read.

Comparing the code to the fixed version shown below, the most important difference is that if loading is not finished, the patched version creates new ArrayBuffer objects using the ArrayBuffer::Create function while the vulnerable version simply passes on a reference to the existing ArrayBuffer to the DOMArrayBuffer::Create function. ToArrayBuffer always returns the actual state of the ArrayBuffer being built but since the reading is a asynchronous, it may return the same one under some circumstances.

```
1  DOMArrayBuffer* FileReaderLoader::ArrayBufferResult() {
2    DCHECK_EQ(read_type_, kReadAsArrayBuffer);
3    if (array_buffer_result_)
4      return array_buffer_result_;
5
6    // If the loading is not started or an error occurs, r
7    if (!raw_data_ || error_code_ != FileErrorCode::kOK)
8      return nullptr;
9
10   if (!finished_loading_) {
11     return DOMArrayBuffer::Create(
12         ArrayBuffer::Create(raw_data_->Data(), raw_data_
13   }
14
15   array_buffer_result_ = DOMArrayBuffer::Create(raw_data
16   AdjustReportedMemoryUsageToV8(-1 *
17                                 static_cast<int64_t>(raw
18   raw_data_.reset();
19   return array_buffer_result_;
20 }
```

What are those circumstances? The *raw_data_* variable in the code is of the type *ArrayBufferBuilder*, which is used to construct

the result ArrayBuffer from the incrementally read data by dynamically allocating larger and larger underlying ArrayBuffers as needed. The ToArrayBuffer method returns a smart pointer to this underlying ArrayBuffer if the contents read so far fully occupy the currently allocated buffer and creates a new one via slicing if the buffer is not fully used yet.

```
scoped_refptr<ArrayBuffer> ArrayBufferBuilder::ToArrayBu
  // Fully used. Return m_buffer as-is.
  if (buffer_->ByteLength() == bytes_used_)
    return buffer_;

  return buffer_->Slice(0, bytes_used_);
}
```

One way to abuse the multiple references to the same ArrayBuffer is by detaching the ArrayBuffer through one and using the other, now dangling, reference. The javascript *postMessage()* method can be used to send messages to a JS Worker. It also has an additional parameter, *transfer*, which is an array of *Transferable* objects, the ownership of which are transfered to the Worker.

The transfer is done by the *blink::SerializedScriptValue::TransferArrayBufferContents* function, which iterates over the *DOMArrayBuffers* provided in the *transfer* parameter to *postMessage* and invokes the *Transfer* method of each, as shown below. *blink::-*

August 2013

January 2013

December 2012

November 2012

September 2012

August 2012

June 2012

CATEGORIES

1Day

exploitation

internet explorer

NDay

News

DOMArrayBuffer::Transfer calls into *WTF::ArrayBuffer::Transfer*, which transfers the ownership of the underlying data buffer.

The vulnerability can be triggered by passing multiple *blink::DOMArrayBuffers* that reference the same underlying ArrayBuffer to *postMessage*. Transferring the first will take ownership of its buffer, then the transfer of the second will fail because its underlying ArrayBuffer has already been neutered. This causes *blink::SerializedScriptValue::TransferArrayBufferContents* to enter an error path, freeing the already transferred ArrayBuffer but leaving a dangling reference to it in the second *blink::DOMArrayBuffer*, which can then be used to access the freed memory through JavaScript.

```
1  SerializedScriptValue::TransferArrayBufferContents(
2  ...
3  for (auto* it = array_buffers.begin(); it != array_buffe
4    DOMArrayBufferBase* array_buffer_base = *it;
5    if (visited.Contains(array_buffer_base))
6      continue;
7    visited.insert(array_buffer_base);
8
9    wtf_size_t index = static_cast<wtf_size_t>(std::distar
10   ...
11   DOMArrayBuffer* array_buffer = static_cast<DOMArrayBuf
12
13   if (!array_buffer->Transfer(isolate, contents.at(index
14     exception_state.ThrowDOMException(DOMExceptionCode::
15       "ArrayBuffer at index " +
16       String::Number(index) +
17       " could not be transferred.");
18     return ArrayBufferContentsArray();
19   }
20 }
```


Other

training

Uncategorized

Vulnerabilities

Exploitation

The vulnerability can be turned into an arbitrary read/write primitive by reclaiming the memory region pointed to by the dangling pointer with JavaScript TypedArrays and corrupting their length and backing store pointers. This can then be further utilized to achieve arbitrary code execution in the renderer process.

Memory Management in Chrome

There are several aspects of memory management in Chrome that affect the reliability of the vulnerability. Chrome uses PartitionAlloc to allocate the backing store of ArrayBuffers. This effectively separates ArrayBuffer backing stores from other kinds of allocations, making the vulnerability unexploitable if the region that is freed is below 2MiB in size because PartitionAlloc will never reuse those allocations for other kinds of data. If the backing store size is above 2MiB, it is placed in a directly mapped region. Once freed, other kinds of allocations can reuse such a region. However, successfully reclaiming the freed region is only possible on 32-bit platforms, as PartitionAlloc adds additional randomness to its allocations via VirtualAlloc and mmap address hinting on 64-bit platforms beside their ASLR slides.

On a 32-bit Windows 7 install, the address space of a fresh Chrome process is similar to the one shown below. Note that these addresses are not static and will differ by the ASLR slide of Windows. Bottom-up allocations start from the lower end of the address space, the last one is the reserved region starting at *36681000*. Windows heaps, PartitionAlloc regions, garbage collected heaps of v8 and Chrome, thread stacks are all placed among these regions in a bottom-up fashion. The backing store of the vulnerable *ArrayBuffer* will also reside here. An important thing to note is that Chrome makes a 512MiB reserved allocation (from *4600000* on the listing below) early on. This is done because the address space on x86 Windows systems is tight and gets fragmented quickly, therefore Chrome makes an early reservation to be able to hand it out for large contiguous allocations, like *ArrayBuffers*, if needed. Once an ArrayBuffer allocation fails, Chrome frees this reserved region and tries again. The logic that handles this could complicate exploitation, so the exploit starts out by attempting a large (1GiB) ArrayBuffer allocation. This will cause Chrome to free the reserved region, then fail to allocate again, since the address space cannot have a gap of the requested size. While most OOM conditions kill the renderer process, ArrayBuffer allocation failures are recoverable from JavaScript via exception handling.

```
...
45f5000  45f8000     3000 MEM_PRIVATE MEM_COMMIT  PAGE_R
45f8000  4600000     8000 MEM_PRIVATE MEM_RESERVE
4600000  24600000 20000000 MEM_PRIVATE MEM_RESERVE
24600000 24601000     1000 MEM_PRIVATE MEM_COMMIT  PAGE_
24601000 24602000     1000 MEM_PRIVATE MEM_RESERVE
...
36681000 36690000     f000 MEM_PRIVATE MEM_RESERVE
36690000 65fc0000 2f930000             MEM_FREE    PAGE_
65fc0000 65fc1000     1000 MEM_IMAGE   MEM_COMMIT  PAGE_
65fc1000 66085000    c4000 MEM_IMAGE   MEM_COMMIT  PAGE_
66085000 66086000     1000 MEM_IMAGE   MEM_COMMIT  PAGE_
...
```

Another important factor is the non-deterministic nature of the multiple garbage collectors that are involved in the managed heaps of Chrome. This introduces noise in the address space that is hard to control from JavaScript. Since the *onprogress* events used to trigger the vulnerability are also fired a non-deterministic number of times, and each event causes an allocation, the final location of the vulnerable ArrayBuffer is uncontrollable without the ability to trigger garbage collections on demand from JavaScript. The exploit uses the code shown below to invoke garbage collection. This makes it possible to free the results of *onprogress* events continuously, which helps in avoiding out-of-memory kills of the renderer process and also forces the dangling pointer created upon triggering the vulnerability to point to the lower end of the address space, somewhere into the beginning of the original 512MiB reserved region.

```
function force_gc() {
```

```
  // forces a garbage collection to avoid OOM kills and
  try {
    var failure = new WebAssembly.Memory({initial: 32767
  } catch(e) {
    // console.log(e.message);
  }
}
```

Exploitation steps

The exploit achieves code execution by the following steps:

- Allocate a large (128MiB) string that will be used as the source of the *Blob* passed to *FileReader*. This allocation will end up in the free region following the bottom-up allocations (from *36690000* in the address space listing above).
- Free the 512MiB reserved region via an oversized *ArrayBuffer* allocation, as discussed previously.
- Invoke *FileReader.readAsArrayBuffer*. A number of *onprogress* event will be triggered, the last couple of which can return references to the same underlying ArrayBuffer if the timing of the events is right. This step can be repeated indefinitely until successful without crashing the process.
- Free the backing store of the ArrayBuffer through one of the references. Going forward, another reference can be used to access the dangling pointer.

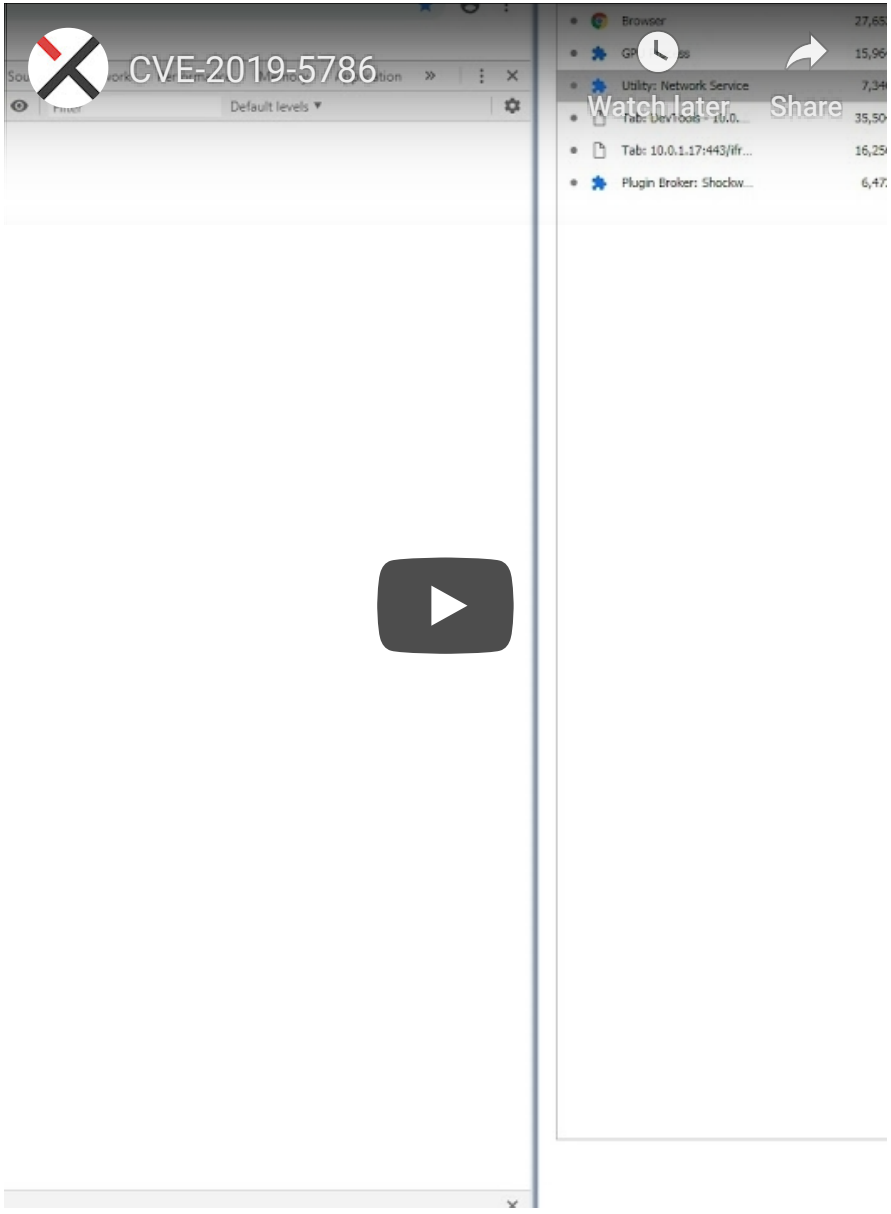
- Reclaim the freed region by spraying the heap with recognizable JavaScript objects, interspersed with TypedArrays.
- Look for the recognizable pattern through the dangling reference. This enables leaking the address of arbitrary objects by setting them as properties on the found object, then reading back the property value through the dangling pointer.
- Corrupt the backing store of a sprayed TypedArray and use it to achieve arbitrary read write access to the address space.
- Load a WebAssembly module. This maps a read-write-executable memory region of 64KiB into the address space.
- Traverse the JSFunction object hierarchy of an exported function from the WebAssembly module using the arbitrary read/write primitive to find the address of the read-write-executable region.
- Replace the code of the WebAssembly function with shellcode and execute it by invoking the function.

Increasing reliability

A single run of the exploit (which uses the steps detailed above) yields a success rate of about 25%, but using a trick you can turn that into effectively 100% reliability. Abusing the site isolation

feature of Chrome enables brute-forcing, as described in another post on this blog by Ki Chan Ahn (look for the section titled “Making a Stealth Exploit by abusing Chrome’s Site Isolation”). A site corresponds to a (scheme:host) tuple, therefore hosting the brute forcing wrapper script on one site which loads the exploit repeatedly in an iframe from another host will cause new processes to be created for each exploit attempt. These iframes can be hidden from the user, resulting in a silent compromise. Using multiple sites to host the exploit code, the process can be parallelized (subject to memory and site-isolation process limits). The exploit developed uses a conservative timeout of 10 seconds for one iteration without parallelization and achieves code execution on average under half a minute.

The entire exploit code can be found on our github and it can be seen in action below.



Detection

The exploit doesn't rely on any uncommon features or cause unusual behavior in the renderer process, which makes distinguishing between malicious and benign code difficult without false positive results.

Mitigation

Disabling JavaScript execution via the Settings / Advanced settings / Privacy and security / Content settings menu provides effective mitigation against the vulnerability.

Conclusion

It's interesting to see exploits in the wild still targeting older platforms like Windows 7 x86. The 32-bit address space is so crowded that additional randomization is disabled in PartitionAlloc and win32k lockdown is only available starting Windows 8. Therefore, the lack of mitigations on Windows 7 that are present in later versions of Windows make it a relatively soft target for exploitation.

Subscribers of our N-Day feed can leverage our in-depth analysis of critical vulnerabilities to defend themselves better, or use the provided exploits during internal penetration tests.

← Exploiting the Magellan bug on 64-bit Chrome Desktop

A window of opportunity: exploiting a Chrome 1day vulnerability →