March 16, 2021

One day short of a full chain: Part 2 - Chrome sandbox escape

Man Yue Mo

In this post I'll go through the exploit of Chrome issue 1125614 (GHSL-2020-165), a bug that I reported in September 2020. This vulnerability will be used for escaping the Chrome sandbox to gain third-party app privilege from a compromised Chrome renderer and as such, I'll assume that I have a compromised renderer throughout the post. The vulnerability affected the `secure-payment-confirmation` feature that was introduced in version 86 of Chrome, which was only in the beta version when I reported it. For the exploit and source code referenced in this post, I'll assume version 86.0.4240.30 of Chrome. As I mistakenly assumed that Chrome on Android is 64 bit, (beta versions around v85 and v86 did come out as 64 bit, although newer versions appeared to have reverted to 32 bit), I'll also assume a 64 bit Chrome binary in this and the next post. This makes the exploit a bit more interesting than it would have been on the 32 bit binary and the technique developed here will also be a bit more general than existing ones.

The vulnerability

The `secure-payment-confirmation` is part of the `Payment API` and can reached by specifying `supportedMethods` to be `secure-payment-confirmation`:

```
var supportedInstruments = [{
                      supportedMethods: 'secure-payment-confirmation',
             ...
                  };
                  var options = {};
          var request = new PaymentRequest(supportedInstruments, details, options);
```

The renderer will check whether the `RuntimeEnabledFeatures::SecurePaymentConfirmation` is enabled before proceeding. When all checks are passed, it'll make an IPC call to the browser using the `PaymentRequest mojom interface`. This will invoke the `PaymentRequest::Init` function in the privileged browser process with the supplied data.

This call will eventually call the native `JNI_PaymentAppServiceBridge_Create` method, which will then call `PaymentAppService::Create` to create the relevant `SecurePaymentConfirmationApp`:

```
void SecurePaymentConfirmationAppFactory::Create(
    base::WeakPtr<Delegate> delegate) {
  ...
  for (const mojom::PaymentMethodDataPtr& method_data : spec->method_data()) {
    if (method_data->supported_method == methods::kSecurePaymentConfirmation) {
      ...
      std::unique_ptr<autofill::InternalAuthenticator> authenticator =
          delegate->CreateInternalAuthenticator();
      ...
}
```

As we passed `secure-payment-confirmation` as the method when creating the `PaymentRequest`, an `InternalAuthenticator` will be created. In this case, it is an `InternalAuthenticatorAndroid` that is created:

```
std::unique_ptr<autofill::InternalAuthenticator>
PaymentAppServiceBridge::CreateInternalAuthenticator() const {
  return std::make_unique<InternalAuthenticatorAndroid>(render_frame_host_);
}
```

As `render_frame_host_` here is stored as a raw pointer in `InternalAuthenticatorAndroid`, the question is then whether `InternalAuthenticatorAndroid` can outlive `render_frame_host_`.

Now `render_frame_host_` here is the `RenderFrameHost` that is tied to the javascript frame where the `PaymentRequest` is made, so by closing the frame, I can cause `render_frame_host_` to be destroyed. So let's take a look at the `InternalAuthenticatorAndroid` that is created:

```
     std::unique_ptr<autofill::InternalAuthenticator> authenticator =
          delegate->CreateInternalAuthenticator();

      authenticator->IsUserVerifyingPlatformAuthenticatorAvailable(
          base::BindOnce(&SecurePaymentConfirmationAppFactory::
                             OnIsUserVerifyingPlatformAuthenticatorAvailable,
                         weak_ptr_factory_.GetWeakPtr(), delegate,
                         method_data->secure_payment_confirmation.Clone(),
                         std::move(authenticator)));                          //<------- passed as argument to `OnIsUserVerifyingPlatformAut
      return;
```

The `authenticator` is created as a `unique_ptr` and is not owned by any other object. It is then passed to `SecurePaymentConfirmationAppFactory::OnIsUserVerifyingPlatformAuthenticatorAvailable` as a callback from `InternalAuthenticator::IsUserVerifyingPlatformAuthenticatorAvailable`. The `InternalAuthenticatorAndroid::IsUserVerifyingPlatformAuthenticatorAvailable` will put this callback into the `is_uvpaa_callback_` field:

```
void InternalAuthenticatorAndroid::
    IsUserVerifyingPlatformAuthenticatorAvailable(
        blink::mojom::Authenticator::
            IsUserVerifyingPlatformAuthenticatorAvailableCallback callback) {
```

```
  ...
  is_uvpaa_callback_ = std::move(callback);
  Java_AuthenticatorImpl_isUserVerifyingPlatformAuthenticatorAvailableBridge(
      env, obj);
}
```

This then calls the `isUserVerifyingPlatformAuthenticatorAvailableBridge` of the `AuthenticatorImpl` in Java:

```
    public void isUserVerifyingPlatformAuthenticatorAvailableBridge() {
        ...
        isUserVerifyingPlatformAuthenticatorAvailable(
                (isUVPAA)
                        -> AuthenticatorImplJni.get()
                                .invokeIsUserVerifyingPlatformAuthenticatorAvailableResponse(
                                        mNativeInternalAuthenticatorAndroid, isUVPAA));
    }
```

This then calls `isUserVerifyingPlatformAuthenticatorAvailable` and calls `invokeIsUserVerifyingPlatformAuthenticatorAvailableResponse` in the callback, which is when `is_uvpaa_callback_` is called:

```
void InternalAuthenticatorAndroid::
    InvokeIsUserVerifyingPlatformAuthenticatorAvailableResponse(
        JNIEnv* env,
        jboolean is_uvpaa) {
  std::move(is_uvpaa_callback_).Run(static_cast<bool>(is_uvpaa));
}
```

As the `unique_ptr` of `authenticator` is moved to `is_uvpaa_callback_`, at the very least, `authenticator` will be kept alive until `is_uvpaa_callback_` is invoked. So let's now go back to `isUserVerifyingPlatformAuthenticatorAvailable` to see when the callback will get invoked:

```
   public void isUserVerifyingPlatformAuthenticatorAvailable(
           IsUserVerifyingPlatformAuthenticatorAvailableResponse callback) {
        ...
        mIsUserVerifyingPlatformAuthenticatorAvailableCallbackQueue.add(callback);
        Fido2ApiHandler.getInstance().isUserVerifyingPlatformAuthenticatorAvailable(
                mRenderFrameHost, this);
    }
```

Here `isUserVerifyingPlatformAuthenticatorAvailable` first adds the `callback` to its `mIsUserVerifyingPlatformAuthenticatorAvailableCallbackQueue`, and then it calls `isUserVerifyingPlatformAuthenticatorAvailable` of the `Fido2ApiHandler`, which is an Android platform API. The callback in the `mIsUserVerifyingPlatformAuthenticatorAvailableCallbackQueue` gets called when a response from the `Fido2ApiClient` is received

From this, it appears that `InternalAuthenticatorAndroid` can be kept alive at least until the Android API `isUserVerifyingPlatformAuthenticatorAvailable` returns; and as it is an Android platform API, its return time would not be dependent on the lifetime of any `RenderFrameHost` in Chrome. So at least in theory, it may be possible to extend the lifetime of the `InternalAuthenticatorAndroid` and have the `RenderFrameHost` deleted before it.

The next question is how is the `RenderFrameHost` used by `InternalAuthenticatorAndroid`. It turns out that there is only a single "use" of it, which is in the destructor of `InternalAuthenticatorAndroid`:

```
InternalAuthenticatorAndroid::~InternalAuthenticatorAndroid() {
  // This call exists to assert that |render_frame_host_| outlives this object.
  // If this is violated, ASAN should notice.
  DCHECK(render_frame_host_);
  render_frame_host_->GetRoutingID();
}
```

and it is calling the virtual `GetRoutingID`. So ironically, the code that is meant for detecting memory corruption problems gave me a very certain and easy to exploit primitive. I only need to make `InternalAuthenticatorAndroid` to outlive `render_frame_host_` and I'll get a certain call to a virtual function, which should be very easy to exploit as a sandbox escape on Android.

Staying alive and winning a race

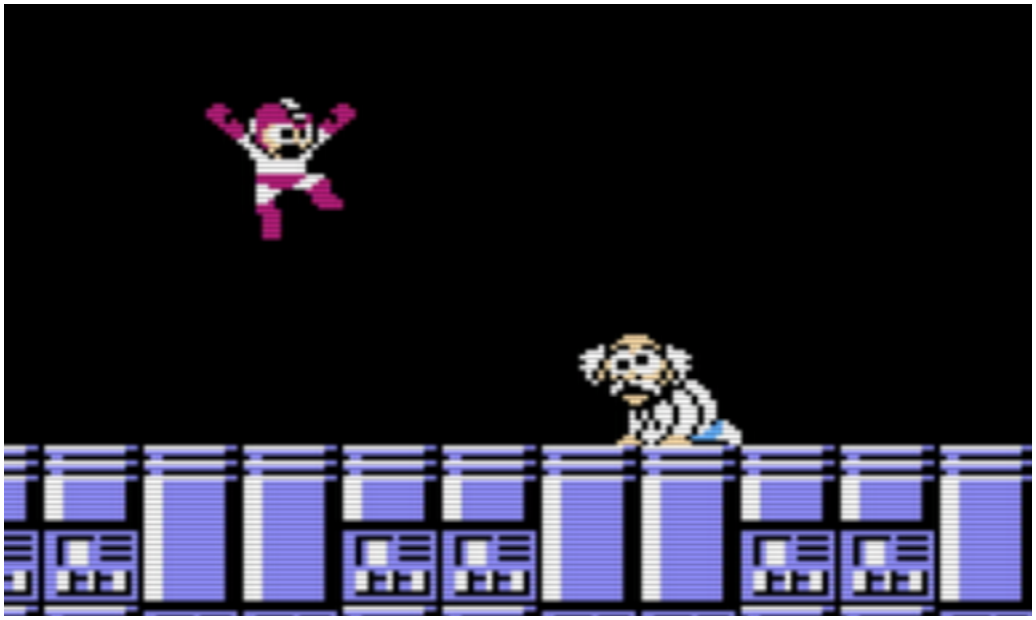
In order to keep the `authenticator` that is created here alive for long enough, observe that its live time is dependent upon the `isUserVerifyingPlatformAuthenticatorAvailable` call returning. As this is a call to an Android service, it is shared by all other frames, (and in fact, all other apps), so by making a lot of calls to this service from different frames, it may be possible to cause it to slow down and buy us the time needed to delete the `render_frame_host_` owned by `authenticator`. One possibility is to create many `PaymentRequest` with the `secure-payment-confirmation` method in the another frame, so that many calls to `isUserVerifyingPlatformAuthenticatorAvailable` will be made. This, unfortunately, will upset the Java garbage collector and cause an out-of-memory error most of the time, so a more lightweight way of jamming the `Fido2ApiClient` is needed. To achieve this, I use the `PublicKeyCredential.isUserVerifyingPlatformAuthenticatorAvailable` function in the `Web Authentication API` to achieve this. By making many calls to this function, it is possible to cause up to a few seconds of delay in the destruction of `authenticator`, enough to trigger the bug multiple times and replace the free'd `render_frame_host_`.

This method, however, would only work on phones with production images, because the `WEB_AUTH` feature is absent on emulator and phones with development OS build. It is fair to say that this is the case most of the time.

Another subtlety is that the `render_frame_host_` cannot be deleted too soon, otherwise a null pointer dereference will crash the browser when it is used to obtain the `WebContents` here. However, with the big time window that I was able to create, this was not a problem at all.

Now that I am able to win the race and trigger the bug, as well as have enough time to replace the free'd `render_frame_host_`, it should just be a straightforward port of Cleanly Escaping the Chrome Sandbox by Tim Becker.

However, it turns out that there are a few problems.



Library address randomization and Zygote

It is well known that all Android processes are forked from the Zygote process, and so all libraries loaded in Zygote have the same base address in all Android processes. In particular, this means that many libraries have the same base addresses in both the renderer and the browser process of Chrome. So once the renderer process is compromised, the base addresses of its libraries can be used to locate gadgets in the browser process as well. This, however, only applies to libraries loaded in Zygote:

```
$ out/86/bin/chrome_public_apk ps
W    0.117s TimeoutThread-1-for-MainThread  Stale cache detected. Not using it.
9A261FFAZ009KQ (aosp_flame-userdebug 10 QQ3A.200805.001 eng.mmo.20210115.132601 test-keys):
org.chromium.chrome 9297
org.chromium.chrome:privileged_process0 9366
org.chromium.chrome:sandboxed_process0:org.chromium.content.app.SandboxedProcessService0:0 9352

flame:/ # cat /proc/9366/maps | grep libhwui
70a838c000-70a8512000 r--p 00000000 fd:00 2420                           /system/lib64/libhwui.so
70a8512000-70a89a3000 --xp 00186000 fd:00 2420                           /system/lib64/libhwui.so
70a89a3000-70a89a4000 rw-p 00617000 fd:00 2420                           /system/lib64/libhwui.so
70a89a4000-70a89cb000 r--p 00618000 fd:00 2420                           /system/lib64/libhwui.so
flame:/ # cat /proc/9352/maps | grep libhwui
70a838c000-70a8512000 r--p 00000000 fd:00 2420                           /system/lib64/libhwui.so
70a8512000-70a89a3000 r-xp 00186000 fd:00 2420                           /system/lib64/libhwui.so
70a89a3000-70a89a4000 rw-p 00617000 fd:00 2420                           /system/lib64/libhwui.so
70a89a4000-70a89cb000 r--p 00618000 fd:00 2420                           /system/lib64/libhwui.so
```

As we can see, the system library `libhwui.so` is loaded in the same location in both the renderer (9352) and the browser (9366) processes. The same applies to many other system libraries that are loaded in Zygote. However, as `libchrome` is not a library loaded in Zygote, its address is still randomized between the renderer and the browser:

```
flame:/ # cat /proc/9352/maps | grep chrome
6fb851d000-6fb8654000 r--s 00943000 fd:05 7311                           /data/app/org.chromium.chrome-FJs_E5QD8BjRukkPI7RdDQ==/base.apk
...
701ed8e000-701ed9b000 r--s 00a79000 fd:05 7311                           /data/app/org.chromium.chrome-FJs_E5QD8BjRukkPI7RdDQ==/base.apk
flame:/ # cat /proc/9366/maps | grep chrome
6fb6293000-6fb68c7000 r--s 00310000 fd:05 7311                           /data/app/org.chromium.chrome-FJs_E5QD8BjRukkPI7RdDQ==/base.apk
...
701e764000-701e78e000 r--s 07a6c000 fd:05 7311                           /data/app/org.chromium.chrome-FJs_E5QD8BjRukkPI7RdDQ==/base.apk
```

So knowing the base address of `libchrome.so` in the renderer does not help locating gadgets inside `libchrome` in the browser process. As such, the gadgets in Cleanly Escaping the Chrome Sandbox, which are inside `libchrome.so` cannot be used and new gadgets have to be found elsewhere. That being said, it is a minor issue as there are plenty other libraries to work with and some well-known gadgets as well, such as the `Execute` function of `libwebp` (compiled into `libhwui.so`) that was used in the MMS Exploit by Mateusz Jurczyk (j00ru).

Per thread arena and thread cache of jemalloc

This is a much bigger problem. The usual heap spraying primitive, when exploiting Chrome sandbox escape, is the `BlobRegistry::registerFromStream` method first used in Virtually Unlimited Memory: Escaping the Chrome Sandbox by Mark Brand. It has the advantage of being able to both write arbitrary data to fake an object and to read back the object after some function calls are made with the fake object. This is particularly useful for leaking a heap address once ASLR is (at least partially) defeated, as in the case of sandbox escape. For example, in our case, I can trigger the bug, and then replace the free'd `InternalAuthenticatorAndroid` with controlled data using `BlobRegistry::registerFromStream` and call a different virtual function of this form:

```
void foo() {
  this.bar = new Bar();
}
```

and then read the address of the field `bar` afterwards will give me a heap address, which can then be used for storing fake vtable to allow any function to be called.

The problem with using `BlobRegistry` heap spraying is that it is running on the `IO` thread, whereas the object I need to replace, `RenderFrameHost` are deleted in the `UI` thread. The Chrome browser process runs two threads, while most objects are created in `UI` thread, some IPC calls are handled by the `IO` thread, and `BlobRegistry` is one of those. This makes the `BlobRegistry` heap spraying rather difficult to use on Android, because the memory allocator on Android <= 10.0 is jemalloc, which is optimized for multi-threading. As part of the optimization, each thread allocates memory from a different region (arena) to minimize the need of locking. Moreover, each thread also holds a thread cache, in which newly free'd objects are placed in the thread cache and are not available to other threads. This makes it very difficult to replace the free'd `RenderFrameHost` with `BlobRegistry` heap spraying. While I was able to replace the free'd `RenderFrameHost` with `BlobRegistry` by spraying more of both `RenderFrameHost` and `Blob` objects, the reliability becomes rather poor when trying to win the race condition at the same time, because the `IO` thread also races with the `UI` thread, making the heap spraying even more unreliable.

This means a new heap spraying primitive is needed. While there are many alternatives, not many of them allows the data to be read again, which makes it difficult to leak a heap address.

Calling arbitrary function

On 32 bit binary, a well-known technique first introduced by Guang Gong in An exploit Chain to Remotely Root Modern Android Devices and also used later by Lucas P. in Yet another RenderFrameHostImpl UAF can be used to call `system` in the plt section of `llvm-glnext.so` with controlled argument. The idea is to replace the free'd object so that the first four bytes, which represents the pointer to the vtable, points to the correct offset in the plt section of `llvm-glnext.so`. Then when `GetRoutingID` is called, it will call `system` with the object itself as the argument instead. As pointers are four bytes in 32 bit binaries, the first four bytes of the argument to `system` will just be the pointer to the vtable, which is unlikely to contain any zero. The rest of the object can then be replaced with the actual desired argument for calling `system` to run arbitary commands.

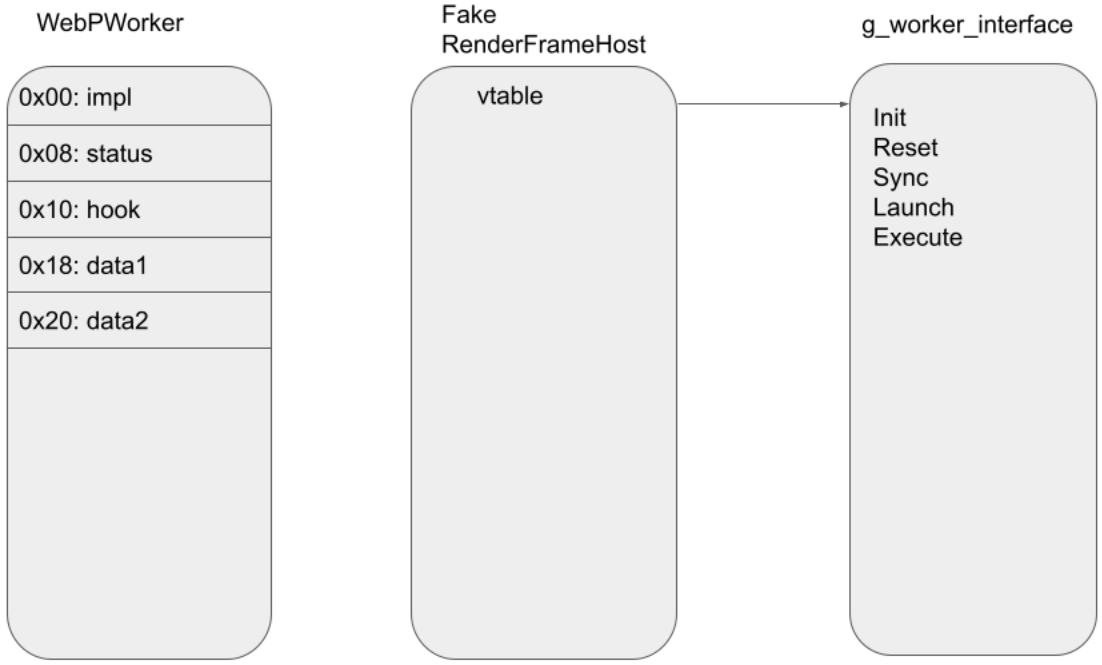
For 64 bit binaries, however, pointers are 64 bit and the actual address only ocuppies 39 bits, which means that the vtable address will certainly contains zero and hence the argument to `system` will be terminated when using the aforementioned technique. In this post, I'll assume 64 bit binary is used and develope the exploit to overcome this hurdle. The method works also for 32 bit binaries, though a bit overkill in that case.

Instead of using `system` from `llvm-glnext.so` directly, I'll use the `Execute` gadget in the library `libhwui.so` mentioned earlier. As pointed out in the MMS Exploit of Mateusz Jurczyk (j00ru), the library `libhwui.so` contains a static `g_worker_interface` variable (part of the statically linked `libwebp`), which stores a pointer to the `Exeucte` function:

```
static void Execute(WebPWorker* const worker) {
  if (worker->hook != NULL) {
    worker->had_error |= !worker->hook(worker->data1, worker->data2);
  }
}
```

By using `g_worker_interface` as a fake vtable, I can create a fake `RenderFrameHost` object that will call `Execute` when the use-after-free bug triggers and its `GetRoutingID` function is called:

```
InternalAuthenticatorAndroid::~InternalAuthenticatorAndroid() {
  ...
  //render_frame_host_ is already free'd
  render_frame_host_->GetRoutingID();
}
```



In this context, the free'd `render_frame_host_` will be treated as a `WebPWorker` object `worker` when `Execute` is called, which means that by faking the fields corresponding to `hook`, `data1` and `data2`, I can call an arbitrary function with arguments that I can control.

However, since `data1` and `data2` are both pointers, to be able to control these arguments, I still need to have a valid heap address where I can place data in. So I still need the ability to leak a heap address first.

Heap spraying with clipboard

To get an idea of some possibilities, I ran a rough, simple CodeQL query on a subset of Chrome built from the `src/content/browser` directory:

```
from Field f
where (f.getType().getName().matches("vector<unsigned%") or
      f.getType().getName().matches("SkBitmap"))
select f, f.getDeclaringType(), f.getLocation()
```

This simply looks for fields that are of type `vector<unsigned int|char>` etc. or `SkBitmap`, which are likely to hold data that I can control. In the end, I decided to use `ScopedClipboardWriter`. This is used by the `ClipboardHost mojom` for copy and pasting into the clipboard and allocates objects in the `UI` thread. A compromised renderer can call the methods of `ClipboardHostImpl` to copy and paste data to the clipboard and then read it back again. However, many of the text base methods, such as `WriteText` would limit the range of the input and output data to valid utf-16 characters, which limits it use. The `WriteImage` function, however, allows creation of arbitrary bitmap, which is good for creating fake objects. Moreover, the data can then be read back after calling the `CommitWrite` function that copies it to the Android clipboard (which will also delete the initial data), and then use the `ReadImage` function to read it back. So this would allow me to potentially leak a heap address.

There is, however, one problem with this. The `Clipboard::WriteImage` accepts a `SkBitmap` with four channels, (RGB and alpha), with data layout as follows:

```
  R|G|B|A|R|G|B|A|...|R|G|B|A|....
```

So the data is grouped into pixels that consists of four bytes, with the first being the R(ed) channel, second the G(reen) channel, third the B(lue) channel, and the last the A(lpha) channel.

The `SkBitmap` also needs to have one of the three `alphaType`: `kOpaque_SkAlphaType`, which will cause the alpha channel to be set to 255, `kPremul_SkAlphaType` will first divide the alpha channel by 255 to obtain a float between 0 and 1, and then multiply the other channels data by this value, whereas `kUnpremul_SkAlphaType` does the opposite. This processing will take place when the bitmap is read back from the clipboard and the data that's got written is still raw data.

So for example, if the raw data for the `SkBitmap` is the following:

```
121|122|123|128|....|10|20|30|40|....
```

Then with `kOpaque_SkAlpha`, I got this when I read from the clipboard:

```
121|122|123|255|....|10|20|30|255|....
```

With `kPremul_SkAlphaType`, I'll get

```
60|61|61|128|....|1|3|4|40|....
```

and `kUnpremul_SkAlphaType` will give me

```
242|244|246|128|....|63|127|191|40|....
```

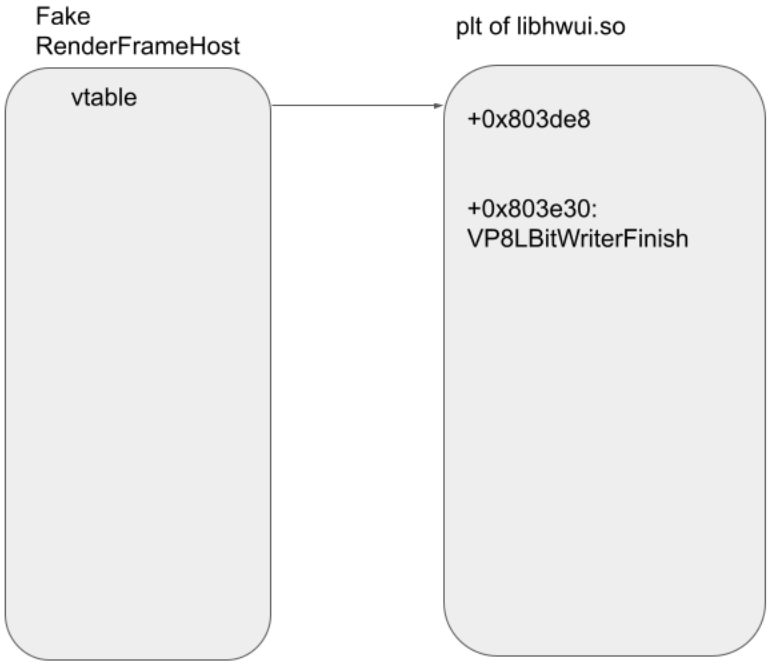
So I can either read the accurate `RGB` values and lose the `Alpha` value or vice versa. As we shall see later, the problem can be overcome by leaking two heap addresses that are close to each other and use bitmaps of different `AlphaType` to leak their addresses. By combining the leaked addresses, an accurate address can be obtained.

An info leak gadget

As explained in Calling arbitrary function above, even if I can call arbitrary (two-argument) functions, to be able to control the arguments, I still need to leak a heap address to some controlled data. To do this, I'll use the `VP8LBitWriterFinish` function, which has a pointer to it stored in the `plt` section of `libhwui.so`.

```
000000803e30  0fb400000402 R_AARCH64_JUMP_SL 00000000007c876c VP8LBitWriterFinish + 0
000000803e38  0b4d00000402 R_AARCH64_JUMP_SL 00000000007c8440 VP8LBitWriterWipeOut + 0
000000803e40  052c00000402 R_AARCH64_JUMP_SL 00000000007c8140 VP8BitWriterAppend + 0
000000803e48  0a4e00000402 R_AARCH64_JUMP_SL 00000000007b1568 VP8LEncDspInit + 0
```

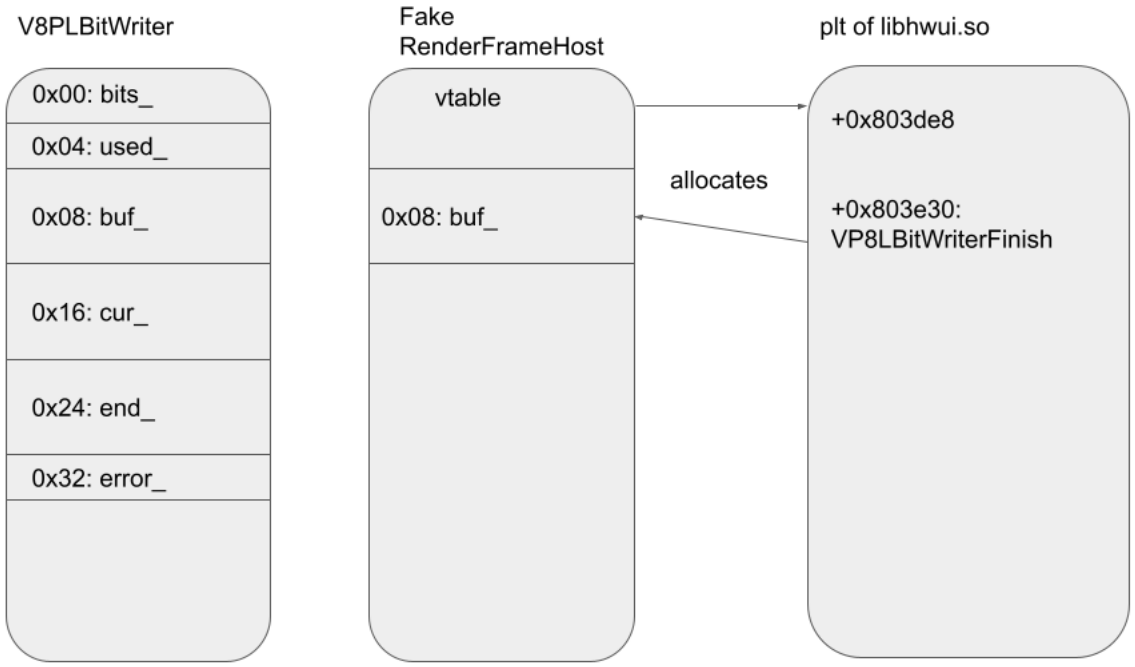
By treating the `plt` section of `libhwui.so` as a gigantic "vtable" and `VP8LBitWriterFinish` as a "virtual function", I can point the vtable of my fake `RenderFrameHost` to an offset in the `plt` section of `libhwui.so` so that when calling `GetRoutingID`, `VP8LBitWriterFinish` will be called instead.



The function `VP8LBitWriterFinish` will call `VP8LBitWriterResize`, which will allocate a new buffer depending on the size of the existing buffer and the `extra_size` argument:

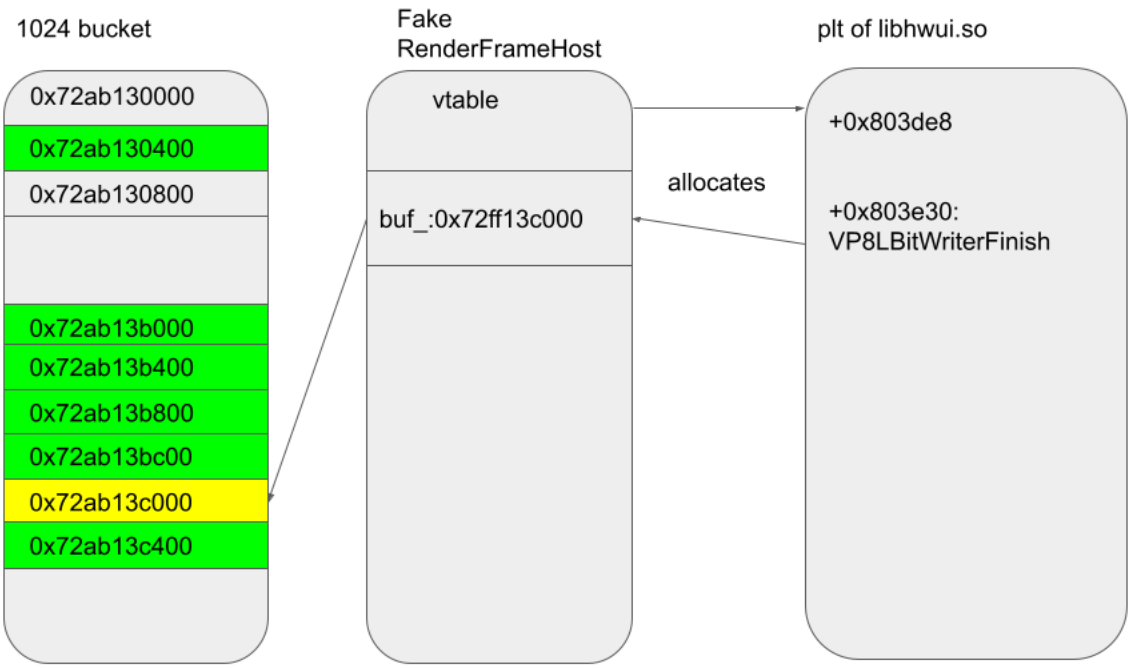
```
static int VP8LBitWriterResize(VP8LBitWriter* const bw, size_t extra_size) {
  uint8_t* allocated_buf;
  size_t allocated_size;
  const size_t max_bytes = bw->end_ - bw->buf_;
  const size_t current_size = bw->cur_ - bw->buf_;
  const uint64_t size_required_64b = (uint64_t)current_size + extra_size;
  const size_t size_required = (size_t)size_required_64b;
  if (size_required != size_required_64b) {
    bw->error_ = 1;
    return 0;
  }
  if (max_bytes > 0 && size_required <= max_bytes) return 1;
  allocated_size = (3 * max_bytes) >> 1;
  if (allocated_size < size_required) allocated_size = size_required;
  // make allocated size multiple of 1k
  allocated_size = (((allocated_size >> 10) + 1) << 10);           //<------ minimal allocation size is 1k
  allocated_buf = (uint8_t*)WebPSafeMalloc(1ULL, allocated_size);  //<------ allocates new buffer if needed, WebPSafeMalloc simply wraps mall
  ...
  WebPSafeFree(bw->buf_);
  bw->buf_ = allocated_buf;              //<---------- stores allocated buffer as a field
  bw->cur_ = bw->buf_ + current_size;
  bw->end_ = bw->buf_ + allocated_size;
  return 1;
}
```

By faking the vtable of the free'd `RenderFrameHost` to call `VP8LBitWriterFinish`, the fake `RenderFrameHost` will be treated as a `VP8LBitWriter`. If I simply leave the rest of the fake `RenderFrameHost` empty (zero), a buffer of size 1k will be created and a pointer to it stored as `bw->buf_`, which is at offset `0x08`:

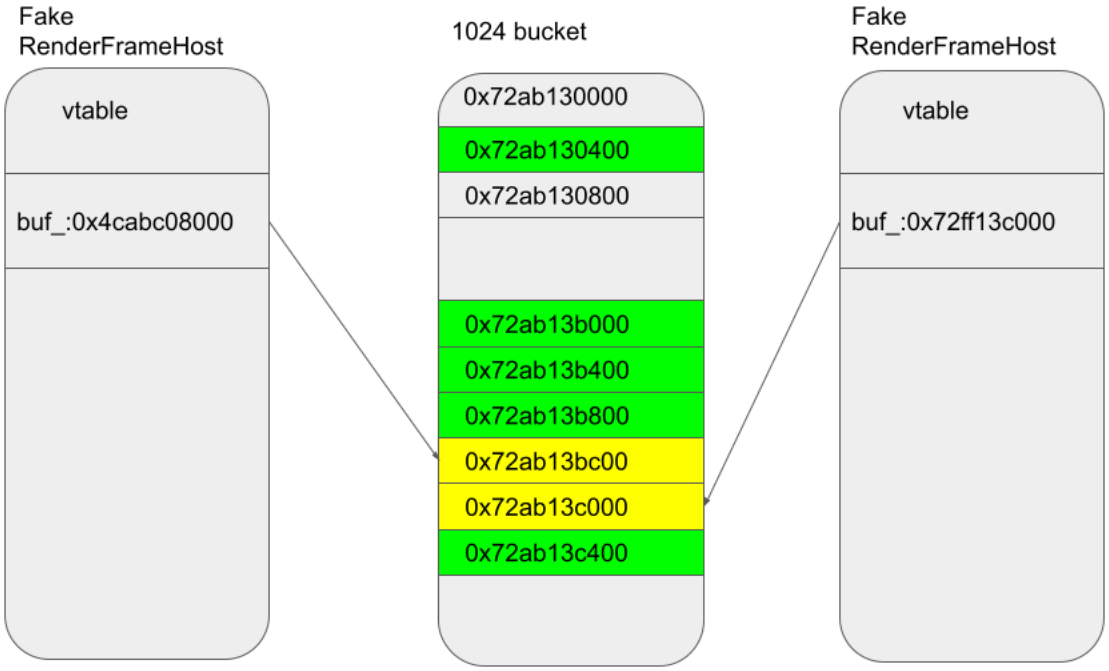


This not only allows us to read the address of this newly created buffer when using the `ClipboardHostImpl::ReadImage` function, but also gives us a large buffer that is in a relatively quiet bucket (size 1024).

Now to overcome the problem mentioned in Heap spraying with clipboard, I can first spray the size 1024 bucket using `ClipboardHostImpl::WriteImage` to fill out the holes, so that allocations in that bucket becomes adjacent to each other. As the bucket is not used often, this can be done with only a small amount of spraying (32 allocation seems to be sufficient). Once that is done, I remove a buffer that is allocated near the end of the heap spray. This leaves a hole in the contiguous buffers that I allocated. I then trigger the bug and replace the free'd `RenderFrameHost` using `ClipboardHostImpl::WriteImage` with a `kOpaque_SkAlphaType` `alphaType` and use the `VP8LBitWriterFinish` to allocate a size 1024 buffer. This is likely to fill the hole that I have created.



However, as also explained in Heap spraying with clipboard, because I'm using `kOpaque_SkAlphaType` as the `alphaType`, the byte that corresponds to the alpha channel, which is the fourth byte, will be masked by `ff`, as shown in the figure in the address of `buf_`. In order to get the full address, I now remove the buffer next to the one that is now occupied by `buf_` and trigger the bug again, but this time with an `alphaType` `kPremul_SkAlphaType`. This time, the address of `buf_` will have the correct fourth byte but all the others incorrect:



As the fourth byte between adjacent buffers are likely to be the same, by combining the addresses, I can get the correct address of the first buffer. This allows me to place data in guessable addresses. In fact, because `buf_` is allocated by `malloc`, it'll even contain the data that I used when I sprayed the 1024 size bucket.

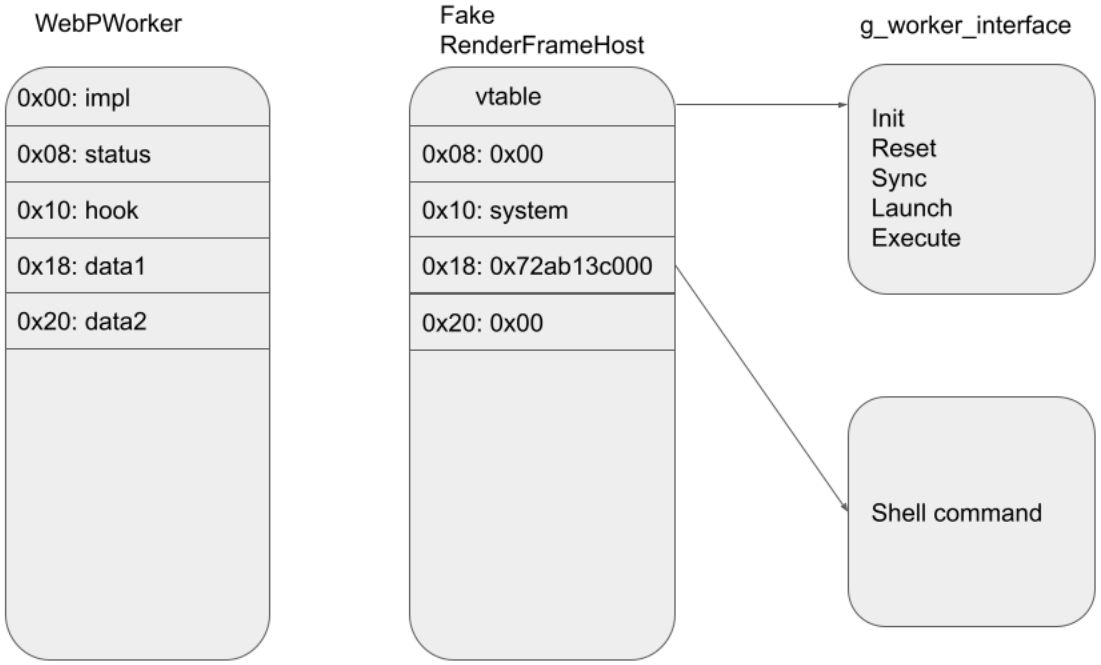
Running arbitrary shell command

Now that I can place data at guessable addresses, I can fully control the arguments that I use when calling functions using `Execute` of `WebPWorker`. I can simply call `system` of `libc.so` to run arbitrary shell commands, as `libc.so` is also one of the standard libraries that is loaded in Zygote:

```
$ out/86/bin/chrome_public_apk ps
W    0.117s TimeoutThread-1-for-MainThread  Stale cache detected. Not using it.
9A261FFAZ009KQ (aosp_flame-userdebug 10 QQ3A.200805.001 eng.mmo.20210115.132601 test-keys):
org.chromium.chrome 9297
org.chromium.chrome:privileged_process0 9366
org.chromium.chrome:sandboxed_process0:org.chromium.content.app.SandboxedProcessService0:0 9352

flame:/ # cat /proc/9366/maps | grep libc.so
70aab66000-70aaba6000 r--p 00000000 07:02 107                            /apex/com.android.runtime/lib64/bionic/libc.so
70aaba6000-70aac4d000 --xp 00040000 07:02 107                            /apex/com.android.runtime/lib64/bionic/libc.so
70aac4d000-70aac50000 rw-p 000e7000 07:02 107                            /apex/com.android.runtime/lib64/bionic/libc.so
70aac50000-70aac57000 r--p 000ea000 07:02 107                            /apex/com.android.runtime/lib64/bionic/libc.so
flame:/ # cat /proc/9352/maps | grep libc.so
70aab66000-70aaba6000 r--p 00000000 07:02 107                            /apex/com.android.runtime/lib64/bionic/libc.so
70aaba6000-70aac4d000 r-xp 00040000 07:02 107                            /apex/com.android.runtime/lib64/bionic/libc.so
70aac4d000-70aac50000 rw-p 000e7000 07:02 107                            /apex/com.android.runtime/lib64/bionic/libc.so
70aac50000-70aac57000 r--p 000ea000 07:02 107                            /apex/com.android.runtime/lib64/bionic/libc.so
```

This will allow me to run any shell command as the Chrome browser process.



The full exploit can be found here with some set up notes.

Conclusions

In this post, I've gone through the details of the sandbox escape in the beta version 86.0.4240.30 of Chrome and we saw that, although the usual heap spraying method that uses `BlobRegistry` is not applicable here, there are still other alternatives that I was able to use to put controlled data into a fake object. We also see that while the base address of Chrome is randomized between the renderer and the browser process, there are still many other gadgets that are available in the preloaded libraries in Zygote. In the end, it didn't take a lot of effort to find appropriate gadgets in those libraries to both leak a heap address and to run arbitrary function. In particular, the one-per-boot-ASLR of Zygote remains a weakness in the Android operating system that greatly reduces the effectiveness of application sandboxing.