Exploiting Chrome IPC

Ned Williamson

November 9, 2018

Self-Employed Researcher, soon Google

Introduction

Who am I?

- 2005: Gamehacking Gunz Online with OllyDbg
- 2011: NDS homebrew user (Tetris TGM3!)
- 2014: CTF with PPP
- 2016: 3DS "jailbreak" dev (Soundhax)
- 2018: Chrome sandbox research
- 2019+: iOS jailbreak (eta not available)

Agenda

- Introducing Generative Coverage-Guided Fuzzing
- Understanding Chrome Sandbox IPC
- Finding Bugs
- Writing an AppCache Fuzzer
- Exploitation

Spot the Bug

```
diff --git a/content/browser/appcache/appcache_group.cc ...
void AppCacheGroup::RemoveCache(AppCache* cache) {
  DCHECK(cache->associated_hosts().empty());
  if (cache == newest_complete_cache_) {
+     CancelUpdate();
    AppCache* tmp_cache = newest_complete_cache_;
    newest_complete_cache_ = NULL;
    tmp_cache->set_owning_group(NULL);
}
```

Almost 3 years ago, this patch was made to Chrome to fix a null dereference. This code was written by a staff engineer with 28 years of experience. It was approved in a code review by an engineer with 17 years of experience. Can you spot the bug that they didn't?

Spot the Bug

- Trick question
- Can't see bug without more context
- Skilled engineers + state of the art engineering process insufficient
- We need advanced testing strategies!

Coverage-Guided Generative Fuzzing

- A new strategy for coverage-guided fuzzing
- Fuzz complex systems efficiently
- Leverage great public tools while still using your brain

Why should I pay attention?

- Syzkaller: $>$ 3000 bugs in the Linux+Android kernels
- Lokihardt: $>$ 40 bugs in Chakra, V8
- Me: $>$ 30 bugs in Chrome sandbox (incl. DoS), 5 critical
- Out of $143k in Chrome bounties, $137k with this technique
- Chrome Bugs– starting to adapt this strategy already!

Why should I pay attention?

- I never had access to scale and didn't want to pay for it
- I found several bugs on my laptop, then later a desktop Core i7
- I never fuzzed more than 8-12 hours at a time without modifying my fuzzer

Fuzzing Primer

Coverage-Guided Fuzzing

- Pioneered late 2014 by lcamtuf with AFL
- Implement function: (bytes $\rightarrow \alpha$), let compiler turn this into (bytes $\rightarrow$ feedback)
- Runtime: randomly mutate bytes in current input corpus
- Save inputs with new feedback
- Using basic block edge coverage as the feedback is effective here
- I personally use libFuzzer today, AFL still good

Coverage-Guided Fuzzing

- Trivial parser functions mostly fuzzed to death
- Exceptions exist: CVE-2018-5146 in libvorbis by fluroescence, a.k.a. mini-Loki
- IPC/DOM/JS fuzzing still done without feedback (generative and/or mutational)
- An obvious next step is then to add coverage feedback to generative fuzzing

Writing a Generative+Coverage-Guided Fuzzer: Easy!

- Write a generative fuzzer natively
- Flip the clang switch to turn on coverage-guided fuzzing
- clang -fsanitize=address,fuzzer

Protobuf

- Google's data serialization format
- Can create a custom input "message" for your fuzz target
- This message format can be structured, think OCaml/Haskell types
- Use protobuf to generate parser for you

Protobuf: Quick Sample Preview

```
enum HttpCode {
  RESPONSE_100 = 100;
  RESPONSE_200 = 200;
  RESPONSE_206 = 206;
  ...
}

message DoRequest { // response from server
  required HttpCode http_code = 1;    // status code, affects AppCache logic
  required bool do_not_cache = 2;     // http headers can indicate no caching
  repeated Url manifest_response = 3; // tells browser which files to request
  required Url url = 4;               // which URL we're serving
}
```

Protobuf-Mutator

- Efficiently generates and mutates your custom protobuf types
- Before, we could only fuzz bytes $\rightarrow$ feedback, maybe writing a hacky byte parser
- With protobuf, we fuzz $\beta \rightarrow$ feedback
- We don't have to fuzz through parser protobuf layer
- Summary: you write a simple grammar and use it; libfuzzer+protobuf-mutator does the rest

Fuzzing Tips

- The more lightweight, the better: less noise, better performance
- The more change in coverage per change in input byte/protobuf tree, the better
- Iterate often: 30-50+ fuzzer versions is not unreasonable
- Edit the target code to make certain scenarios more likely

Chrome IPC Fundamentals

Choosing IPC

- I like privilege escalations starting from arbitrary code
- On 3DS I originally only planned on userland $\rightarrow$ kernel
- Win32k escapes killed in 2016 renderer lockdown
- Great opportunity to practice privilege escalation in Chrome IPC

How Chrome IPC works: Source Layout

- Each tab is in its own sandboxed process
- All tabs communicate to the browser process (outside the sandbox) over IPC
- Each renderer process managed in browser (privileged) process by RenderProcessHost object
- Key folders: src/content/browser, src/net/, really anything src/*/browser
- You can bug hunt IPC parsing/logic, I chose to look at application code

The Attack Surface: Browser Content

- Renderer → Browser IPC mostly ends up in src/content/browser
- src/net was good (critical bugs!) but a net sandbox is coming
- Protip: Look at the lines in RenderProcessHostImpl::Init:
  - CreateMessageFilters(); // old-style
  - RegisterMojoInterfaces(); // new-style
- You can see all the renderer-accessible interfaces installed by these

Old-Style IPC

- Uses “message filters”
- Mostly removed at this point
- Still some code being migrated

Old-Style IPC Example

```
bool RenderFrameMessageFilter::OnMessageReceived(
    const IPC::Message& message) {
  bool handled = true;
  IPC_BEGIN_MESSAGE_MAP(RenderFrameMessageFilter, message)
    IPC_MESSAGE_HANDLER(FrameHostMsg_CreateChildFrame,
                        OnCreateChildFrame)
    IPC_MESSAGE_HANDLER(FrameHostMsg_CookiesEnabled, OnCookiesEnabled)
    IPC_MESSAGE_HANDLER(FrameHostMsg_DownloadUrl, OnDownloadUrl)
    IPC_MESSAGE_HANDLER(FrameHostMsg_SaveImageFromDataURL,
                        OnSaveImageFromDataURL)
```

You can call any of the `On*` functions with controlled params.

Mojo

- Better engineered version of IPC
- Uses .mojom files to define an interface
- Build generates headers for this, you subclass to make your implementation
- Can bind to an available interface in-process or out-of-process

Mojo IPC Example

```
// AppCache messages sent from the child process to the browser.
interface AppCacheBackend {
  // Informs the browser of a new appcache host.
  RegisterHost(int32 host_id);
  // Informs the browser of an appcache host being destroyed.
  UnregisterHost(int32 host_id);
...
```

Looking in the mojom file describing this AppCacheBackend interface gives us a hint about the AppCacheBackend interface being an attack surface.

Mojo IPC Example (from RegisterMojoInterfaces)

```
registry->AddInterface(base::BindRepeating(
  &AppCacheDispatcherHost::Create,
  base::Unretained(storage_partition_impl_->GetAppCacheService()),
  GetID()));
```

Here the browser exposes this interface to the renderer and provides a way to create an AppCacheDispatcherHost.

Mojo IPC Example

```
void AppCacheDispatcherHost::Create(
  ChromeAppCacheService* appcache_service, int process_id,
  mojom::AppCacheBackendRequest request) {

  appcache_service->Unbind(process_id);
  appcache_service->Bind(
  std::make_unique<AppCacheDispatcherHost>(appcache_service, process_id),
  std::move(request), process_id);
}
```

The mojom::AppCacheBackendRequest corresponds to the renderer's request to access the interface. The appcache service binds that request to a new dispatcher host, which is owned by the appcache service.

Mojo IPC Example

```
class AppCacheDispatcherHost : public mojom::AppCacheBackend {
 public:
  AppCacheDispatcherHost(ChromeAppCacheService* appcache_service,
                         int process_id);
  ~AppCacheDispatcherHost() override;
 private:
  // mojom::AppCacheHost
  void RegisterHost(int32_t host_id) override;
  void UnregisterHost(int32_t host_id) override;
  void SetSpawningHostId(int32_t host_id, int spawning_host_id) override;
```

Here's the object we want to fuzz! Note that it subclasses mojom::AppCacheBackend.

Fuzzing In-Process

- Libfuzzer+protobuf-mutator must fuzz in-process, not over IPC
- This is easy to handle: Mojo can work transparently in-process vs. out-of-process
- For old-style IPC, just hack around it by calling endpoints directly from C++

Finding Bugs

Process Overview

Now we know what IPC looks like, let's look at my process and how I designed my AppCache fuzzer.

1. Pick your subsystem
2. Pick your targeted files
3. Figure out how user input affects control flow
4. Write your structured fuzzer
5. Evaluate coverage and go back to 4. or give up

Picking your subsystem

- Attacking from two sides: IPC + network, IPC + Disk, etc.
- Previous bug reports: very useful
- Places where nullptr derefs or use after frees are getting fixed (see git logs)
- Look for manual parsing or other dangerous stuff (raw pointers, scoped_refptrs)
- Knowing more patterns from a strong auditing foundation highly recommended

Pick your targeted files

- Write down and visualize the hierarchy of your chosen module
- Think about writing a clean integration test for the interesting files
- Breaking down the fuzzer into the right size is a matter of taste - human intuition!

Figure out where user input affects control flow

In Chrome browser process, these are particularly useful:

- Controlled data read from disk, network, IPC message
- Threading
- Timing of Callbacks

Picking My Subsystem+Targeted Files: Looking at Old Reports

gzobqq is a prolific Chrome researcher.

In late 2015 he reported 3 issues in AppCache that could lead to sandbox escape.

This gives hints about files where bugs might lie dormant and how to trigger them.

These reports are what I would call textbook auditing bugs... until now.

Here are the bugs and the relevant attack surface to trigger them:

- Bug 551044: Compromised Renderer IPC message
- Bug 554908: Compromised Renderer IPC message + Server Response Timing
- Bug 558589: Server HTTP Response Codes (critical!)

AppCache: Attack Surface Summary

The main objects where the bugs were triggered in the previous reports:

- AppCacheHost
- AppCacheUpdateJob
- AppCacheDispatcherHost

The ways to introduce user input:

- Arbitrary IPC Messages
- HTTP Server Response Codes
- Timing of the above w.r.t. async task loop

Spoiler: my sandbox escape uses *all* of these!

AppCache: Reviewing the Class Hierarchy

```
Browser Network Request Loader: SubresourceLoader, AppCacheURLRequest
Browser Frame Host Handle: AppCacheNavigationHandle
Browser Storage
  AppCacheRequestHandler
    AppCacheURLLoaderJob/AppCacheJob
      AppCacheResponse
  ChromeAppCacheService inherits AppCacheServiceImpl
    AppCacheGroup owns AppCacheUpdateJob, AppCache
      AppCacheUpdateJob::UpdateURLLoaderRequest
Renderer Process Host (browser side)
  AppCacheDispatcherHost
    // IPC via AppCacheHost, has pointers to AppCache and AppCacheGroup
```

AppCache: What We Want to Test

```
Browser Network Request Loader: SubresourceLoader, AppCacheURLRequest
Browser Frame Host Handle: AppCacheNavigationHandle
Browser Storage
  AppCacheRequestHandler
    AppCacheURLLoaderJob/AppCacheJob
      AppCacheResponse
  ChromeAppCacheService inherits AppCacheServiceImpl
    AppCacheGroup owns AppCacheUpdateJob , AppCache
      AppCacheUpdateJob::UpdateURLLoaderRequest
Renderer Process Host (browser side)
  AppCacheDispatcherHost
    // IPC via AppCacheHost , has pointers to AppCache and AppCacheGroup
```

From reviewing the code we cover everything we want by making a ChromeAppCacheService and creating one or more AppCacheHosts and using IPC to eventually create an AppCacheUpdateJob.

AppCache: Mocking the Network

- AppCacheURLLoaderJob instantiates jobs from a network job factory
- When creating the high level AppCacheService, substitute a mocked one
- We can serve or pre-cache a response at any time
- Original fuzzer mocked out server response headers using protobuf
- Current fuzzer simplified, only HTTP status codes and cache headers

AppCache: Fuzzer Protobuf Specification

```
message Session {
  repeated Command commands = 1;
}
```

First we write our custom message format. The fundamental fuzz loop will be to complete a sequence of “commands” using the API.

AppCache: Fuzzer Protobuf Specification

```
message Command {
  oneof command {
    RegisterHost register_host = 1;   // Send IPC messages
    UnregisterHost unregister_host = 2;
    SelectCache select_cache = 3;
    SetSpawningHostId set_spawning_host_id = 4;
    SelectCacheForSharedWorker select_cache_for_shared_worker = 5;
    MarkAsForeignEntry mark_as_foreign_entry = 6;
    GetStatus get_status = 7;
    StartUpdate start_update = 8;
    SwapCache swap_cache = 9;
    GetResourceList get_resource_list = 10;
    DoRequest do_request = 11;        // Synthesize server response
    RunUntilIdle run_until_idle = 12; // Run task loop
  }
}
```

AppCache Protobuf: IPC Messages

```
message RegisterHost {
  required HostId host_id = 1;
}
message UnregisterHost {
  required HostId host_id = 1;
}
// can trigger manifest fetch from manifest url
message SelectCache {
  required HostId host_id = 1;
  required HostId from_id = 2;
  required Url document_url = 3;
  required Url opt_manifest_url = 4;
}
```

AppCache Protobuf: Network Mocking

```
enum HttpCode {
  RESPONSE_100 = 100;
  RESPONSE_200 = 200;
  RESPONSE_206 = 206;
  ...
}

message DoRequest { // response from server
  required HttpCode http_code = 1;    // status code, affects AppCache logic
  required bool do_not_cache = 2;     // http headers can indicate no caching
  repeated Url manifest_response = 3; // tells browser which files to request
  required Url url = 4;               // which URL we're serving
}
```

AppCache Protobuf: C++ Fuzz Target

```
DEFINE_BINARY_PROTO_FUZZER(const fuzzing::proto::Session& session) {
  // Initialize appcache service against mocked network, and create
  // one dispatcher host (used to send "IPC" messages).
  network::TestURLLoaderFactory mock_url_loader_factory;
  SingletonEnv().InitializeAppCacheService(&mock_url_loader_factory);
  mojom::AppCacheBackendPtr host;
  AppCacheDispatcherHost::Create(SingletonEnv().appcache_service.get(),
                                 /*process_id=*/1, mojo::MakeRequest(&host));
  // fuzzer main loop
  for (const fuzzing::proto::Command& command : session.commands()) {
    ...
```

The DEFINE_BINARY_PROTO_FUZZER macro works with libprotobuf-mutator. We simply tell it we want to be fed Session typed messages, and it does the rest.

AppCache Protobuf: C++ Fuzz Target

```
void DoRequest(...) {
  factory->SimulateResponseForPendingRequest(
      url, status,
      response_head /* http_code, do_not_cache */,
      response_body /* manifest_response */);
}
```

We use a mocked network backend to service network requests. This will respond to blocked network request or preload a response that will complete instantly when the url is requested. Now all possible response timings are encapsulated here deterministically.

Note that I'm reusing unit test code - thanks Google!

AppCache Protobuf: C++ Fuzz Target

```
for (const fuzzing::proto::Command& command : session.commands()) {
  switch (command.command_case()) {
    case fuzzing::proto::Command::kRegisterHost: {
      host->RegisterHost(command.register_host().host_id());
      break;
    }
    ...
    case fuzzing::proto::Command::kDoRequest: {
      DoRequest(&mock_url_loader_factory,
                command.do_request().url(), command.do_request().http_code(),
                command.do_request().do_not_cache(),
                command.do_request().manifest_response());
      break;
    ...
```

We either make IPC calls or do network requests. Easy!

CVE-2018-17462: Sandbox escape in AppCache, $80000 get

```
==9269==ERROR: AddressSanitizer: heap-use-after-free
READ of size 4 at 0x60f0000ab3a0 thread T0
    #0 0x90884cd in Release
    #3 0x90884cd in ~scoped_refptr
    #4 0x90884cd in content::AppCacheHost::~AppCacheHost()
    #8 0x905d7fe in ~unique_ptr
    #15 0x905d7fe in content::AppCacheBackendImpl::~AppCacheBackendImpl()
    #16 0x9074dd7 in ~AppCacheDispatcherHost

0x60f0000ab3a0 is located 0 bytes inside of 176-byte region
freed by thread T0 here:
    #0 0x31b8842 in operator delete(void*)
    #6 0x9088331 in content::AppCacheHost::~AppCacheHost()
    #10 0x9148937 in ~unique_ptr
    #20 0x9075a24 in UnregisterHost
    #21 0x9075a24 in content::AppCacheDispatcherHost::UnregisterHost(int)
```

This Fuzzer is Open Source!

To see the full fuzzer in context, download chromium source and view src/content/browser/appcache/appcache_fuzzer.{cc,proto}

Another Example: Network Disk Cache

- Network cache is in the browser process (unsandboxed)
- Hand-optimized for performance, prone to bugs
- The cache allows you to create/destroy cache entries by key
- You can read and write to a cache entry at different offsets
- All of this reachable from JS by seeking on a remote media file

Disk Cache Example: Protobuf Format

```
// edited for simplicity
message Session {
  repeated Command commands = 1;
  int32 max_size = 2; // set session-wide settings
}
message Command {
  oneof command {
    CreateEntry create_entry = 1;           // name
    WriteSparseData write_sparse_data = 2; // name, offset, num bytes
    CloseEntry close_entry = 3;             // name
  }
}
```

Disk Cache Example: The Generated Test

```
TEST_F(DiskCacheBackendTest, SparseEvict) {
  SetMaxSize(512); InitCache();
  scoped_refptr<net::IOBuffer> buffer(new net::IOBuffer(64));
  disk_cache::Entry* entry0 = nullptr, entry1 = nullptr, entry2 = nullptr;
  CreateEntry("http://www.0.com/", &entry0);
  CreateEntry("http://www.1.com/", &entry1);
  CreateEntry("http://www.15360.com/", &entry2);
  WriteSparseData(entry0, 0, buffer.get(), 64);
  WriteSparseData(entry0, 67108923, buffer.get(), 1);
  WriteSparseData(entry1, 53, buffer.get(), 1);
  WriteSparseData(entry2, 0, buffer.get(), 1);
  entry1->Close();
  entry2->Close();
  entry0->Close();
}
```

Disk Cache Example: CVE-2018-6085

```
==6480==ERROR: AddressSanitizer: heap-use-after-free
READ of size 8 at 0x611000bca4c0 thread T6 (CacheThread_Blo)
    #0 0x7b9ca3f in disk_cache::SparseControl::WriteSparseData()
    #1 0x7b9c535 in disk_cache::SparseControl::~SparseControl()
    #2 0x7b7a76e in operator()
    #3 0x7b7a76e in reset
    #4 0x7b7a76e in disk_cache::EntryImpl::~EntryImpl()
0x611000bca4c0 is located 0 bytes inside of 240-byte region
freed by thread T6 (CacheThread_Blo) here:
    #0 0x25ad352 in operator delete(void*)
    #1 0x7b56bed in scoped_refptr<disk_cache::EntryImpl>::operator=(...)
    #2 0x7b5b0b0 in disk_cache::BackendImpl::InternalDoomEntry(...)
    #3 0x7b814f2 in disk_cache::Eviction::EvictEntry(...)
    #4 0x7b7eb3e in disk_cache::Eviction::TrimCache(bool)
    #5 0x7b7abcc in disk_cache::EntryImpl::~EntryImpl()
```

Disk Cache Example: CVE-2018-6085

```
commit df5b1e1f88e013bc96107cc52c4a4f33a8238444
Author: Maks Orlovich <morlovich@chromium.org>
Date:   Fri Mar 30 03:51:06 2018 +0000

    Blockfile cache: fix long-standing sparse + evict reentrancy problem

    Thanks to nedwilliamson@ (on gmail) for an alternative perspective
    plus a reduction to make fixing this much easier.

    Bug: 826626, 518908, 537063, 802886
```

Disk Cache Example: CVE-2018-6085

- 518908 is the oldest bug referenced closed by this fix
- Bugs are filed sequentially on the Chrome tracker
- This bug is restricted, but we find 518912 is open
- This crash happened in the wild on or before Aug 10 2015
- Fixed March 2018... 2 years, 7 months later
- March 2018 = pwn2own, had held this bug for several months beforehand
- 5 minute old profile needed to trigger bug, assumed it wouldn't qualify
- Google had the crash report for this bug and could not solve it in 3 years

Disk Cache Fuzzing: 3 Years Unsolved vs. 1 Day to Find

- One Saturday night, browsed cs.chromium.org and found this attack surface
- Sunday morning, wrote the fuzzer that found 3 *critical* bugs in a couple hours
- CVE-2018-6085, CVE-2018-6086, CVE-2018-6118

Exploiting AppCache

Exploiting AppCache

The focus of this talk is bug finding, but I'd like to share some exploit details.

What is the bug?

The AppCache bug lets you reentrantly add a reference to a destroyed AppCache object multiple times. You can later destroy those objects holding the extra references at any time to trigger decrefs on the dangling AppCache pointer.

At this point, Niklas Baumstark joined me to write the exploit. From my fuzzer we could work side-by-side the crashing input to replicate it on ASAN Chrome. Now we have our initial setup: it's time to exploit!

The Primitive: Decref-by-N-After-Free

Decref decrements the reference counter in the object, then does the following:

- If refcount is still > 0, return.
- If refcount is now 0, call object destructor and free the object.

This means we can safely decrement the first dword of an object N times as long as it doesn't become 0. The AppCache destructor has a virtual delete, meaning we can do a controlled vtable call.

Basics

To exploit a memory corruption bug, you generally need:

- A memory disclosure bug
- Instruction pointer control

Luckily Windows randomizes per-module per-boot, so because we owned the renderer we already know where our ROP gadgets are! We still need a heap pointer so we will need to use our decref to cause a leak somehow.

How do we leak?

- Reclaim the block using an object that we can read from renderer
- net::CanonicalCookie is same LFH size class and readable over IPC!

net::CanonicalCookie

```
class NET_EXPORT CanonicalCookie {
  std::string name_; // <- I target this pointer, which we can read over IPC
  std::string value_;
  std::string domain_;
  std::string path_;
  base::Time creation_date_;
  base::Time expiry_date_;
  base::Time last_access_date_;
  bool secure_;
  bool httponly_;
  CookieSameSite same_site_;
  CookiePriority priority_;
}
```

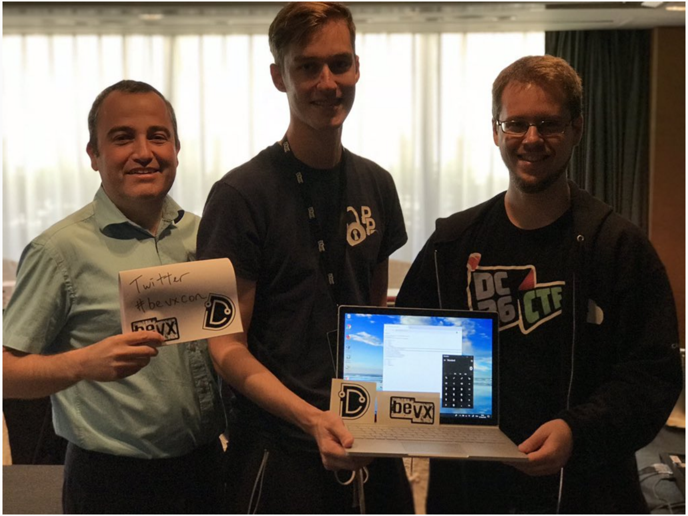
The Leak

- Trigger the bug to get dangling references; spray cookies via normal HTTP request
- Trigger the decref multiple times to corrupt renderer-accessible string pointer
- Read cookie back over IPC to see leaked name
- The browser sends the raw name bytes back and the renderer turns to Unicode
- Requires renderer hack to disable Unicode parsing.. no problem, thanks niklasaelo

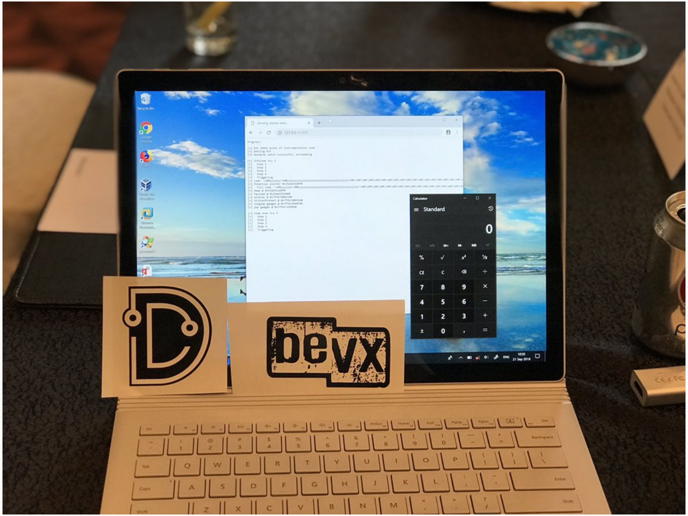
How do we control RIP?

- Make fake AppCacheGroup with refcount 0
- Make fake AppCache with a refcount 1, point to AppCacheGroup (using heap leak + spray)
- Destroy AppCache → destroys AppCacheGroup
- AppCacheGroup dtor has vtable call, and using leak we have controlled data
- Bootstrap ROP using longjmp, done
- Thanks to Niklas for Windows skills on this part!
- Thanks to @5aelo for his RCE to enable the full chain demo
- He should have an upcoming talk that focuses more on the explotiation details ;)

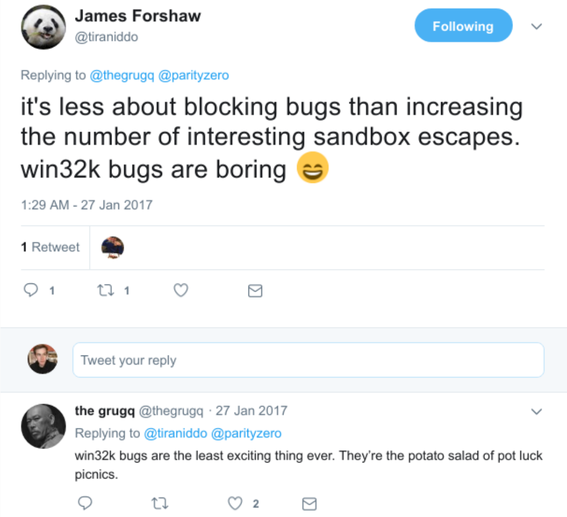
Success (Thank You Beyond Security!)



Success (Thank You Beyond Security!)



:)


James Forshaw
@tiraniddo
Following
Replying to @thegrugq @parityzero
it's less about blocking bugs than increasing the number of interesting sandbox escapes. win32k bugs are boring 😄
1:29 AM - 27 Jan 2017
1 Retweet
1
1
Tweet your reply
the grugq @thegrugq · 27 Jan 2017
Replying to @tiraniddo @parityzero
win32k bugs are the least exciting thing ever. They're the potato salad of pot luck picnics.
2

Lessons

- Chrome Windows browser process lacks CFI, exploitation not too bad
- Windows has LFH... good lord that is annoying - but still exploitable
- Windows choice to make text predictable across modules made this exploitable
- IPC exploits are still completely doable in 2018
- Thank you to the win32k lockdown for letting my show my strength

Demo?

Questions?