Controlled chaos: Predicting object addresses in Chrome (without breaking a sweat)

Who am I?

Man Yue Mo

- Works at GitHub Security Lab, focus on Chrome and Android security
- Most work can be found at the GitHub Security Lab website: https://securitylab.github.com/research/ and at GitHub blog: https://github.blog/author/mymo/

What this talk is about

- Getting compressed (lower 32 bits) addresses of objects in V8, both builtin objects, (e.g. object maps) and user allocated objects
- Getting addresses of executable region (e.g. compiled JIT code)
- “Bruteforcing” the top 32 bit address

Use cases

- Bugs with write primitive or UAF that allows calling of virtual function. For example, exploiting UAF in blink usually need to know addresses to controlled data (e.g. address of user allocated V8 object) and executable memory region (e.g. jump to rop gadgets etc.)
- Less often to have V8 bug that doesn't also give info leak primitive
- So need to know both compressed address and top 32 bits

Historic context

- Side channel attack: Pre spectre [1]

  ASLR on the line: Practical Cache Attacks on the MMU [2]

Historic context

- Side channel attack: Spectre and meltdown

Historic context

- Side channel attack: Spectre and meltdown:

  "Our research reached the conclusion that, in principle, untrusted code can read a process's entire address space using Spectre and side channels."

  -- A year with Spectre: a V8 perspective: https://v8.dev/blog/spectre

Historic context

- Bypassing ASLR using Oilpan's conservative garbage collector [3]

Historic context

We've thought about this and have decided to WontFix this bug, even though it's real … While this is a new avenue, and particularly convenient, we already have to plan for a world in which ASLR is bypassable. (Bummer!)” [4]

Present context

- Nothing new, can be achieved via Spectre/Meltdown
- But can be done a lot simpler
- Comically simple, in fact

Chaos

"… branch of mathematics focused on underlying patterns and deterministic laws, of dynamical systems, that are highly sensitive to initial conditions, that were once thought to have completely random states of disorder and irregularities."

-- Wikipedia "Chaos theory"

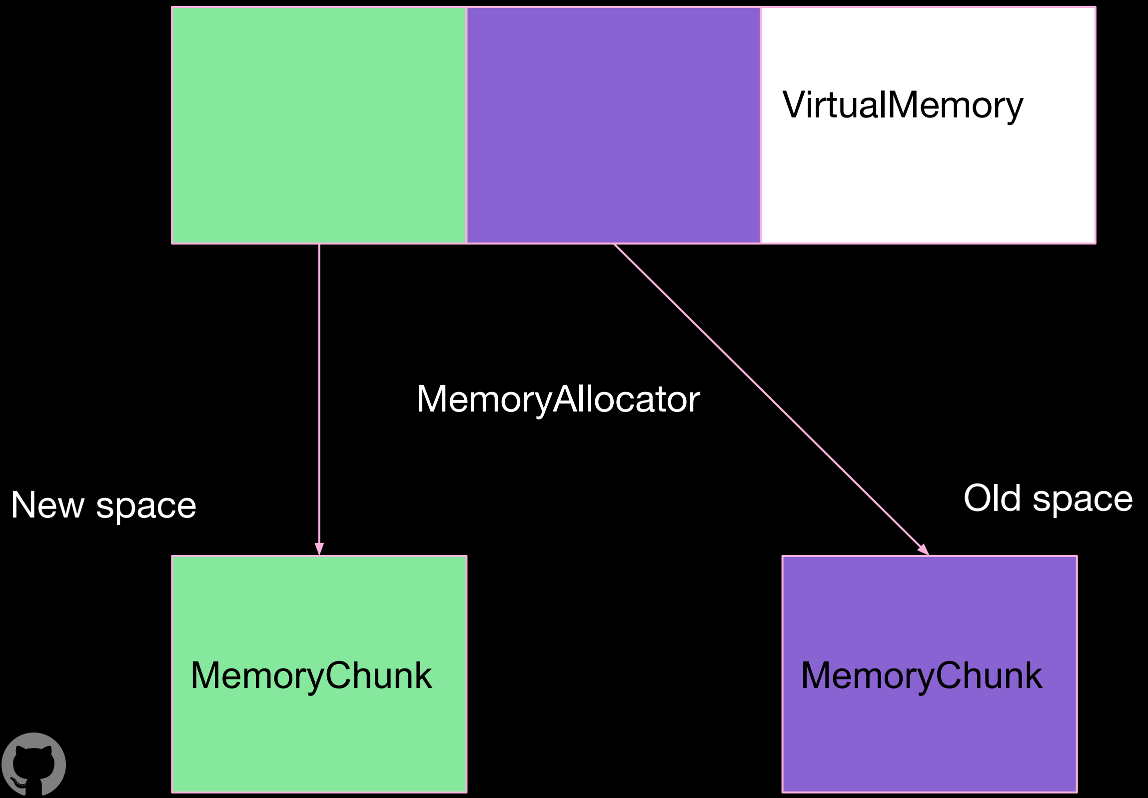
What is the V8 Heap

- Area of 4GB virtual memory (VirtualMemory)
- Aligned to 4GB, i.e. all lower 32 bits zero at the start of area
- Memory allocator use for allocating the memory (MemoryAllocator)
- MemoryAllocator is the interface to allocate memory from the heap

Structure of V8 Heap

- Divided into different spaces, e.g. New space, Old space
- Spaces use the MemoryAllocator of heap to allocate backing stores
- Backing stores are allocated as MemoryChunk
- MemoryChunk contains metadata such as chunk header.
- Spaces are created in Heap::SetupSpaces when the renderer process is initialized

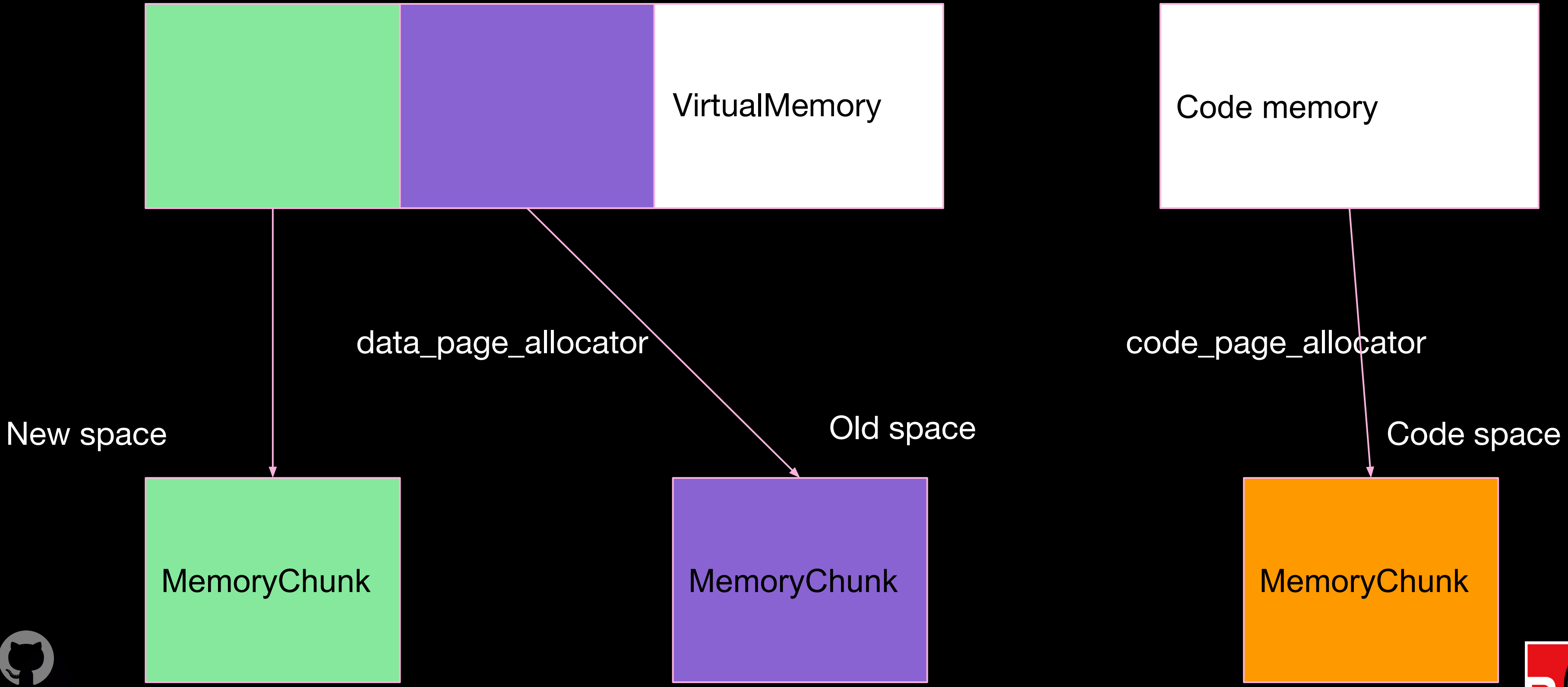
Structure of V8 Heap



Data vs Code

- Two different kinds of spaces: data (non executable) and code (executable)
- MemoryAllocator contains two PageAllocator: data_page_allocator and code_page_allocator
- data_page_allocator allocates data (non executable) pages from the VirtualMemory of the heap
- code_page_allocator allocates code memory (executable) from *outside* the heap

Structure of V8 Heap



Data spaces

- NEW_SPACE: Most newly allocated Objects
- OLD_SPACE: New objects moved to old space after garbage collection, but some objects, like WasmInstance allocated in OLD_SPACE right away
- NEW/OLD_LO(LargetObject)_SPACE: Objects larger than certain threshold (kMaxRegularHeapObjectSize)
- MAP_SPACE: maps for objects

Data spaces

- Different behaviour for allocating backing stores
- New space is a SemiSpaceNewSpace and allocates backing store at creating time
- Other spaces derived directly from PageSpace or Space and allocate backing store on the first object allocation

SemiSpaceNewSpace

- Backing store allocated as MemoryChunk at construction time
- Constructor calls SemiSpace::Commit to allocate the backing store

SemiSpaceNewSpace

```
Page* new_page =
heap()->memory_allocator()->AllocatePage(

        MemoryAllocator::AllocationMode::kUsePool, this,
NOT_EXECUTABLE);

    ...

memory_chunk_list_.PushBack(new_page);
```

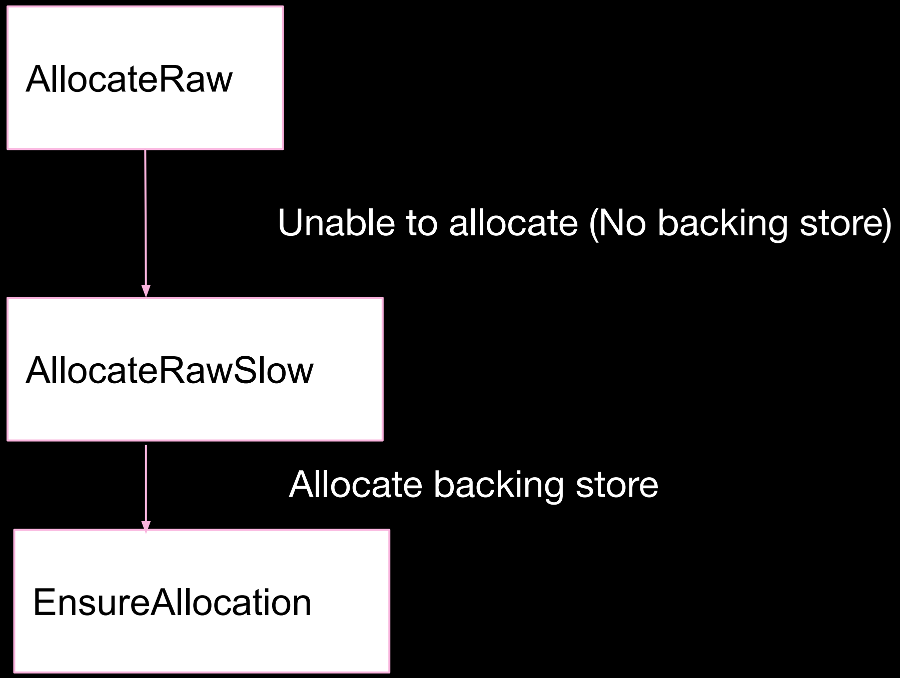
SemiSpaceNewSpace

- Page (Subclass of MemoryChunk) allocated using MemoryAllocator of heap, then added to memory_chunk_list_
- memory_chunk_list_ used for setting current_page_ in SemiSpace::Reset, which is where objects are allocated
- The very first space to be allocated

Old space and Map space

- Backing stores not allocated when spaces are created
- Instead, allocated when object is allocated

Old space and Map space



Old space and Map space

- During initialization of renderer process, built-in objects are allocated in both Old space and Map space, so their backing stores are still allocated during initialization

Large object space

- Similar to Old space and Map space, backing store is only allocated when object is allocated
- No large object is allocated during initialization, so backing store only allocated in Javascript (i.e. we control when it is allocated)

Initialization order

- Heap and spaces are created in Isolate::Init when renderer is created
- Heap::SetUp: Allocates the VirtualMemory region for both the heap and the Code space and creates MemoryAllocator
- Heap::SetUpSpaces: Create the data spaces, the backing store of NEW_SPACE allocated here
- DeserializeIntoIsolate: Deserialize heap snapshot to create built-in objects, maps etc. OLD_SPACE and MAP_SPACE backing store allocated here

MemoryAllocator

- Use data_page_allocator_ and code_page_allocator_ to allocate backing stores
- data_page_allocator_ allocates from heap (4GB VirtualMemory region)
- code_page_allocator_ allocates from a separate region
- Order of allocation is fixed
- What kind of randomness is involved?

MemoryChunk allocation

- MemoryAllocator::AllocateUninitializedChunk:

```
#ifdef V8_COMPRESS_POINTERS
  // When pointer compression is enabled, spaces are expected to be at a
  // predictable address (see mkgrokdump) so we don't supply a hint and rely on
  // the deterministic behaviour of the BoundedPageAllocator.
  void* address_hint = nullptr;
#else
  ...
```

MemoryChunk allocation

- MemoryAllocator::AllocateUninitializedChunk:

```
Address base =
  AllocateAlignedMemory(chunk_size, area_size, MemoryChunk::kAlignment,
                        executable, address_hint, &reservation);
```

address_hint == 0 for compressed pointers =>
Chunk (compressed) addresses not randomized

MemoryChunk allocation

- MemoryAllocator::AllocateAlignedMemory:

```
v8::PageAllocator* page_allocator = this->page_allocator(executable);

VirtualMemory reservation(page_allocator, chunk_size, hint, alignment);
```

page_allocator: data_page_allocator_ or code_page_allocator_ depending on executable

hint: Zero for compressed pointer

page_allocator is a BoundedPageAllocator

MemoryChunk allocation

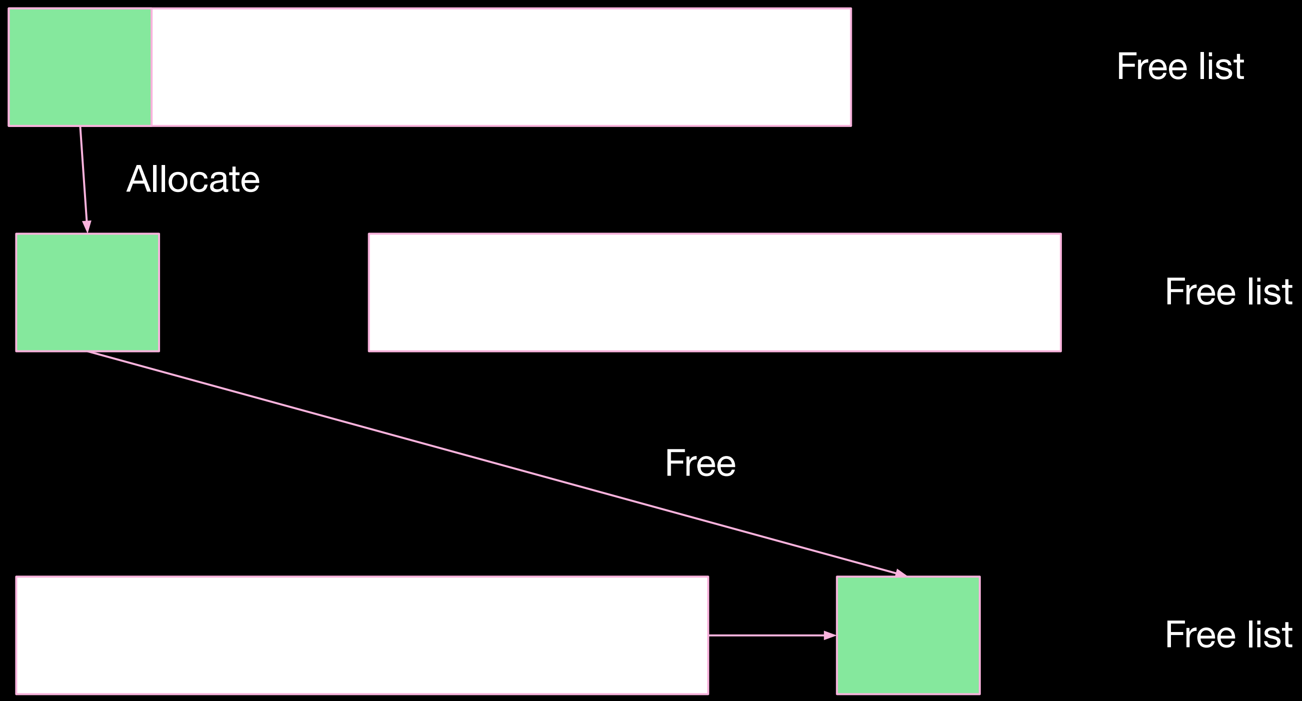
- AllocatePages: Used by VirtualMemory constructor to allocate backing region

```
if (FLAG_randomize_all_allocations) {  //Only for testing
  hint = AlignedAddress(page_allocator->GetRandomMmapAddr(), alignment);
}
void* result = nullptr;
for (int i = 0; i < kAllocationTries; ++i) {          hint == 0
  result = page_allocator->AllocatePages(hint, size, alignment, access);
```

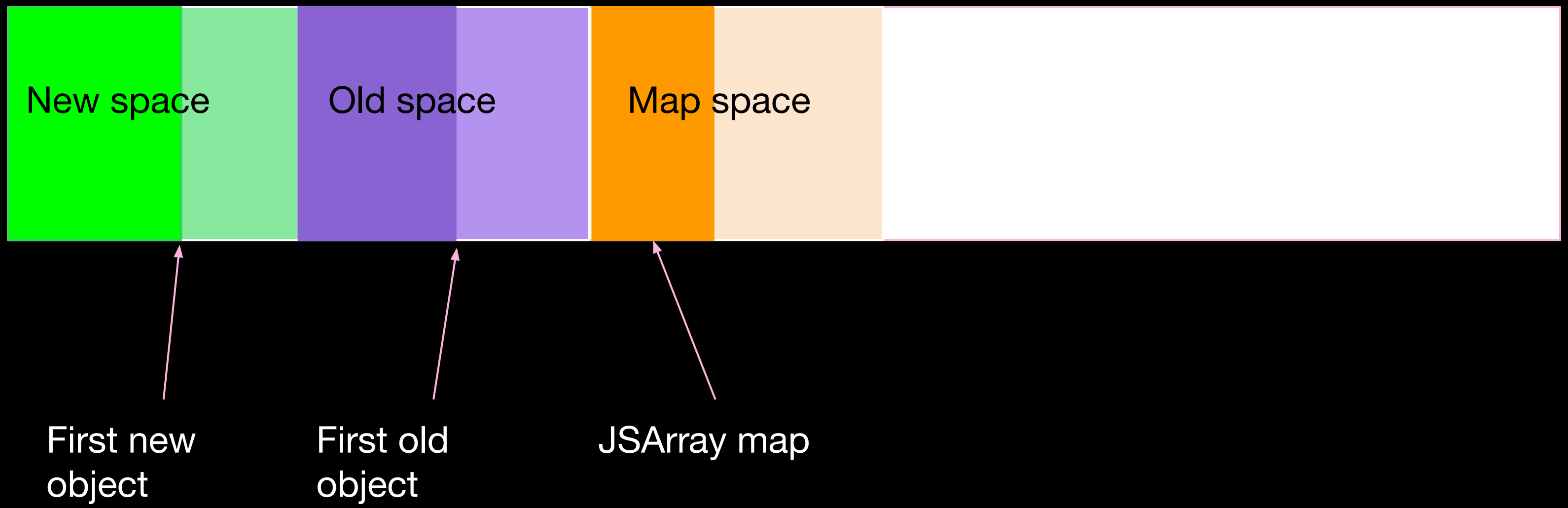
MemoryChunk allocation

- BoundedPageAllocator::AllocatePages: Use RegionAllocator to allocate the pages
- Allocator with a free list that initially consists of the whole region
- When allocation is made, regions in the free list that is large enough is used. The region is split and the remaining region goes back to the free list

RegionAllocator



V8 Heap after initialization



Compressed address

For example, run this JS code (tested 107.0.5304.87):

```
function load() {
    %DebugPrint([1,2]);
}
```

Compressed address

Large Object space

```
DebugPrint: 0x2f410010ac8d: [JSArray]
 - map: 0x2f410024e645 <Map[16](HOLEY_SMI_ELEMENTS)> [FastProperties]
 - prototype: 0x2f410024e101 <JSArray[0]>
 - elements: 0x2f4100282139 <FixedArray[262144]> [HOLEY_SMI_ELEMENTS]
 - length: 262144
 - properties: 0x2f4100002251 <FixedArray[0]>
```

0x280000: New large object space offset

0x2138: Chunk header size

Code space

- Backing store allocated using CodeRange, which is a VirtualMemoryCage separated from the heap
- Integrated into the heap by using the code_page_allocator_
- code_page_allocator_ is a v8::PageAllocator and not a BoundedPageAllocator (i.e. different type from the data_page_allocator_)

Code space

- code_page_allocator_ is not a BoundedPageAllocator but a v8::PageAllocator

```
bool CodeRange::InitReservation(v8::PageAllocator* page_allocator,
                                size_t requested) {
  if (V8_EXTERNAL_CODE_SPACE_BOOL) {
    page_allocator = GetPlatformPageAllocator();
  }
```

Overwrites page_allocator (code_page_allocator_)

Code space

- Backing store allocated during Heap::Setup, at CodeRange::InitReservation

```
params.requested_start_hint =
    GetCodeRangeAddressHint()->GetAddressHint(requested, allocate_page_size);
  if (!VirtualMemoryCage::InitReservation(params)) return false;
```

requested_start_hint used as a hint to allocate backing store

Code space

- GetCodeRangeAddressHint()->GetAddressHint()
- Tries to look for an address near the region where the builtin code is stored

Code space

- Address hint used by code_page_allocator_ to allocate backing store
- code_page_allocator_ (v8::PageAllocator) uses partition_alloc::AllocPages
- Code region size:

```
#elif V8_TARGET_ARCH_ARM64
constexpr size_t kMaximalCodeRangeSize =
    (COMPRESS_POINTERS_BOOL && !V8_EXTERNAL_CODE_SPACE_BOOL) ? 128 * MB
                                                              : 256 * MB;
```

Code space

- partition_alloc::AllocPages:
- Two or Three tries to allocate pages at the hinted address.
- Hinted address updated to a new random address on failure
- Often not enough space to allocate code space near the initial hinted address
- Use random hint most of the time

Code space

- On failure, GetRandomPageBase is used to generate new random hint
- Returns a masked result

```
uintptr_t GetRandomPageBase() {
  …
  random &= internal::ASLRMask();
  random += internal::ASLROffset();
}
```

Code space

- ASLRMask:

  Windows: (1 << 47) (>= windows 8.10)

  (1 << 43) (< windows 8.10)

  MacOS: (1 << 38)

  Linux/ChromeOS: (1 << 46)

Code space

- ASLRMask:

  Android: (1 << 30) (both 64 and 32 bits)

  Code space alignment: (1 << 28) (256 MB), same as Code space size

  4 different possible “random” locations

Code space

For example, run this JS code (Tested 107.0.5304.54):

```
function foo(a, b) { return a + b;}

 function load() {

    %DebugPrint([1,2]);

    //Needs both to allocate JIT code

    for (let i = 0; i < 20000; i++) foo(1, 2);

    x = foo(3, 4);

 }
```

Code space

Top 32 bits

- V8 VirtualMemoryCage reserved in IsolateAllocator::InitializeOncePerProcess
- When heap sandbox is configured, uses sandbox->address_space()->AllocatePages

```
Address base = sandbox->address_space()->AllocatePages(

sandbox->base(), params.reservation_size, params.base_alignment,

PagePermissions::kNoAccess);
```

Top 32 bits

- Uses GetProcessWidePtrComprCage()->InitReservation otherwise

```
if (!GetProcessWidePtrComprCage()->InitReservation(params,
                                                   existing_reservation)) {
```

Top 32 bits

- Either way, uses OS::GetRandomMmapAddr to obtain address hint to map the virtual memory

```
#if V8_TARGET_ARCH_X64 || V8_TARGET_ARCH_ARM64
  raw_addr &= uint64_t{0x3FFFFFFFF000};
```

Address is random and masked to 46 bits. On Arm64, address space is 39 bits, so hint is almost certain to fail and the first free address is used => Fixed once per boot (Memory layout depends on Zygote on Android)

Entropy summary

- Compressed pointer addresses in data space (Object map, JS objects (New, Old, Large): Predictable and deterministic depends on version
- Code space location: “Randomized” to 4 possible locations on Android
- Top 32 bit entropy: 14 bit for x64, once per boot fixed with 1 << 7 = 128 possibilities

Bruteforcing top 32 bits

- OK if only compressed address is needed
- How to get top 32 bits address? (e.g. Use in blink)
- 1 / 128 or even 1 / 4 is not good enough
- Chrome renderer actually is “fault tolerant”

Bruteforcing top 32 bits

- Ki Chan Ahn: Making a Stealth Exploit by abusing Chrome’s Site Isolation [5]
- Site Isolation => Separate renderer process for each different origin
- Create many iframes with different hosts to guess the top 32 bits
- Wrong guess crashes iframe, but does not affect main frame, iframe can be restarted
- Can bruteforce small entropy

Bruteforcing top 32 bits

- Android does not have full site isolation
- Site isolation possible since M92
- By default, different origins shared same process
- Various ways to use site isolation

Cross-Origin-Opener-Policy (COOP) header

- Stated in documentation: https://security.googleblog.com/2021/07/protecting-more-with-site-isolation.html
- Only works for main frame 😔

```
bool NavigationRequest::ShouldRequestSiteIsolationForCOOP() {
  // COOP isolation can only be triggered from main frames.  COOP headers
  // aren't honored in subframes.
  if (!IsInMainFrame()) return false;
```

Sandbox iframe

- Not implemented on Android (as of M 107) 😔

```
bool SiteIsolationPolicy::AreIsolatedSandboxedIframesEnabled() {
  return !IsSiteIsolationDisabled(SiteIsolationMode::kPartialSiteIsolation) &&
         UseDedicatedProcessesForAllSites() &&
         base::FeatureList::IsEnabled(features::kIsolateSandboxedIframes);
}
```

UseDedicatedProcessForAllSites => Full site isolation, not on Android

kIsolateSandBoxedIframes disabled on Android

Origin-Agent-Cluster header

```
bool NavigationRequest::IsOptInIsolationRequested() {

  if (!SiteIsolationPolicy::IsOriginAgentClusterEnabled())

    return false;

  return response_head_->parsed_headers->origin_agent_cluster ==

         network::mojom::OriginAgentClusterValue::kTrue;

}

const base::Feature kOriginIsolationHeader{"OriginIsolationHeader",
base::FEATURE_ENABLED_BY_DEFAULT};
```

Origin-Agent-Cluster header

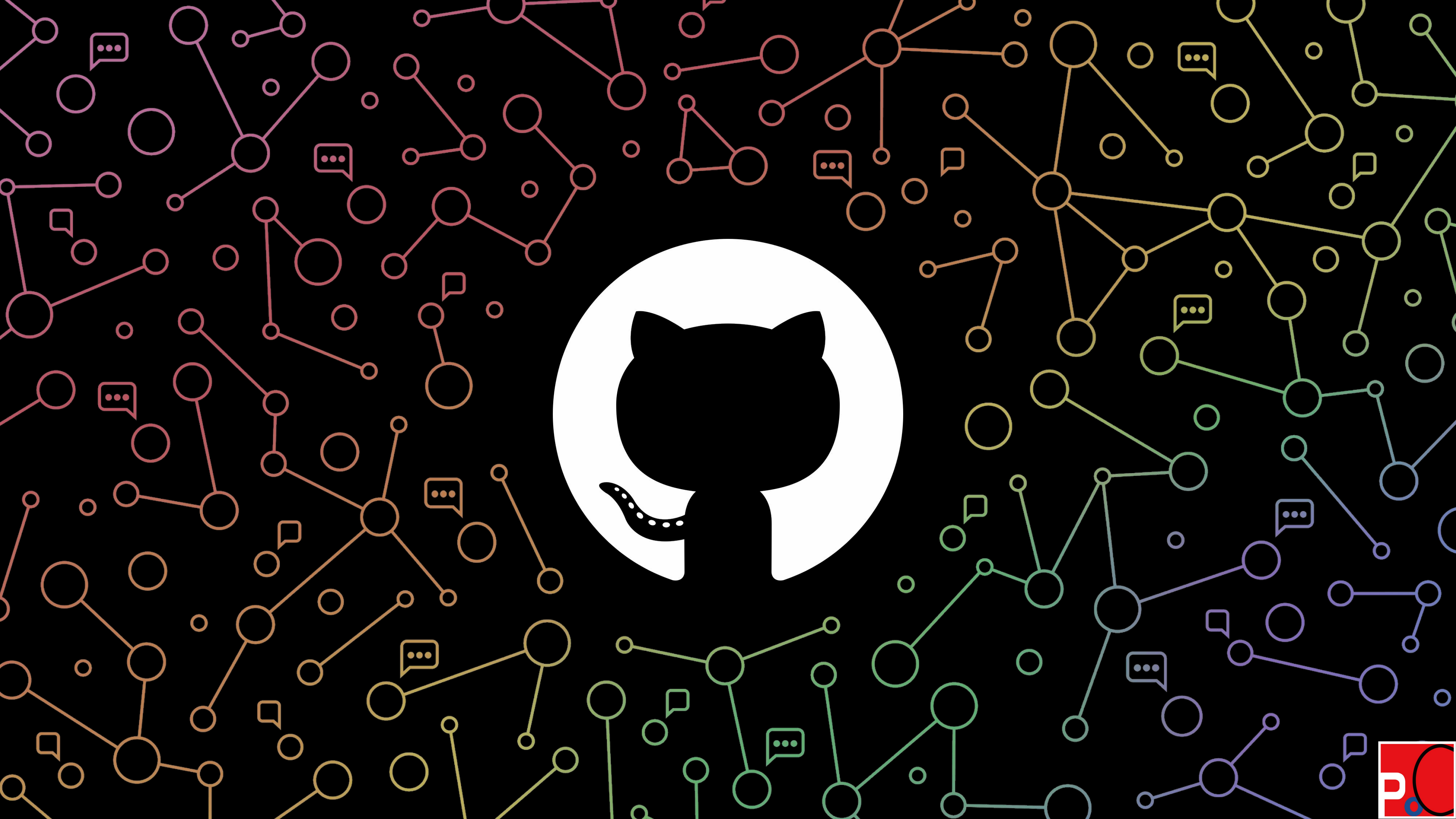
- Feature enabled by default
- Can be used in iframe
- Determined by a header

```
res.writeHead(200, { 'Content-Type': 'text/html',
'Origin-Agent-Cluster': '?1' });
```

Works for both 32 and 64 bit Chrome on Android

Summary

- Compressed addresses (lower 32 bits) are fixed and depends only on Chrome version
- Top 32 bits and Code space can be bruteforce using site isolation
- On Android, code space mostly in 4 locations



References

1. https://bugs.chromium.org/p/chromium/issues/detail?id=665930
2. https://download.vusec.net/papers/anc_ndss17.pdf
3. https://bugs.chromium.org/p/chromium/issues/detail?id=1144662
4. https://bugs.chromium.org/p/chromium/issues/detail?id=1144662#c18

References

5. https://blog.exodusintel.com/2019/01/22/exploiting-the-magellan-bug-on-64-bit-chrome-desktop/?fbclid=IwAR0WiWjsUnun8AuipENIUCMwTvWI35I7rAgsTflQTecmazElNoCAYvm0BsA