State of JS Engine Exploitation

July 2026

Samuel Groß, Google Project Zero

CVE-2019-17026: Zero-Day Vulnerability in Mozilla Firefox Exploited in Targeted Attacks

Security

Google patches type confusion zero-day in Chrome's V8 engine

The bug was discovered by its government-backed attacks focused research group.

The Million Dollar Dissident

NSO Group's iPhone Zero-Days used against a UAE Human Rights Defender

By Bill Marczak and John Scott-Railton August 24, 2016

Responding to Firefox 0-days in the wild

By Philip Martin Corporate, August 8, 2019, 7 min read time

Google Patches Chrome Zero-Day CVE-2025-10585 as Active V8 Exploit Threatens Millions

Sep 18, 2025 Ravie Lakshmanan Vulnerability / Browser Security

CVE-2019-17026: Zero-Day Vulnerability in Mozilla Firefox Exploited in Targeted Attacks

Security

Google patches type confusion zero-day in Chrome's V8 engine

The bug was discovered by its government-backed attacks focused research group.

> 10 Years of JavaScript Engine Exploitation

The Million Dollar Dissident

NSO Group's iPhone Zero-Days used against a UAE Human Rights Defender

By Bill Marczak and John Scott-Railton August 24, 2016

Responding to Firefox 0-days in the wild

By Philip Martin Corporate, August 8, 2019, 7 min read time

Google Patches Chrome Zero-Day CVE-2025-10585 as Active V8 Exploit Threatens Millions

Sep 18, 2025 Ravie Lakshmanan Vulnerability / Browser Security

Outline

1. Why JavaScript engine’s are **hard** to secure
2. Brief history of JavaScript engine exploitation
3. Overview of current and future defenses
4. JS engine vulnerability research in 2026

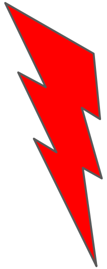
“Classic” Memory Safety Bugs

```
int array[100];
int get(int i) {
    if (i >= 100) return 0;
    return array[i];
}
```

“Classic” Memory Safety Bugs

```c
int array[100];
int get(int i) {
    if (i >= 100) return 0;
    return array[i];
}


get(-1);
```



“Classic” Memory Safety Bugs - Compiler Mitigation

```
int array[100];
int get(int i) {
    if (i >= 100) return 0;
    __bounds_check__(size_t{i}, 100);
    return array[i];
}
```

✓

“Classic” Memory Safety Bugs - Compiler Mitigation

```
int array[100];
int get(size_t i) {
    if (i >= 100) return 0;
    return array[i];
}
```

“Classic” Memory Safety Bugs - Compiler Mitigation

```
int array[100];
int get(size_t i) {
    if (i >= 100) return 0;
    __bounds_check__(size_t{i}, 100);
    return array[i];
}
```

“Classic” Memory Safety Bugs - Compiler Mitigation

```
int array[100];
int get(size_t i) {
    if (i >= 100) return 0;
    __bounds_check__(size_t{i}, 100);
    return array[i];
}
```

Redundant bounds-check will be removed by compiler during code optimization ⇒ Low overhead 🥳

JavaScript Engine Bug

```
let array = new Array(100);
function get(i, m) {
    if (i < 0 ||
        i >= array.length) return;
    return Number(m) * array[i];
}
```

JavaScript Engine Bug

```
let a = new Array(100);
function get(i, m) {
    if (i < 0 ||
        i >= array.length) return;
    return Number(m) * array[i];
}
```

JavaScript Engine Bug

```
let a = new Array(100);
function get(i, m) {
    if (i < 0 ||
        i >= array.length) return;
    return Number(m) * array[i];
}
```

```
FUNC get(i, m):
  SPECULATE typeof(i) == Smi
  IF i < 0 RETURN
  IF i >= array.length RETURN
  M = ToNumber(m)
  __bounds_check__(i, array.length)
  A = array[i]
  RETURN M * A
```

JavaScript Engine Bug

```
let a = new Array(100);
function get(i, m) {
    if (i < 0 ||
        i >= array.length) return;
    return Number(m) * array[i];
}
```

```
FUNC get(i, m):
  SPECULATE typeof(i) == Smi
  IF i < 0 RETURN
  IF i >= array.length RETURN
  M = ToNumber(m)
  __bounds_check__(i, array.length)
  A = array[i]
  RETURN M * A
```

```
FUNC get(i, m):
  SPECULATE typeof(i) == Smi
  IF i < 0 RETURN
  IF i >= array.length RETURN
  M = ToNumber(m)
  __bounds_check__(i, array.length)
  A = array[i]
  RETURN M * A
```

JavaScript Engine Bug

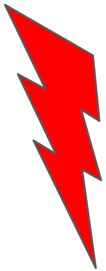
```
let a = new Array(100);
function get(i, m) {
    if (i < 0 ||
        i >= array.length) return;
    return Number(m) * array[i];
}
```

```
FUNC get(i, m):
  SPECULATE typeof(i) == Smi
  IF i < 0 RETURN
  IF i >= array.length RETURN
  M = ToNumber(m)
  __bounds_check__(i, array.length)
  A = array[i]
  RETURN M * A
```

```
FUNC get(i, m):
  SPECULATE typeof(i) == Smi
  IF i < 0 RETURN
  IF i >= array.length RETURN
  M = ToNumber(m)
  __bounds_check__(i, array.length)
  A = array[i]
  RETURN M * A
```

JavaScript Engine Bug

```
let array = new Array(100);
function get(i, m) {
    if (i < 0 ||
        i >= array.length) return;
    return Number(m) * array[i];
}
let evil = { valueOf() { array.length = 0; } };
get(42, evil);
```



JavaScript Engine Bug

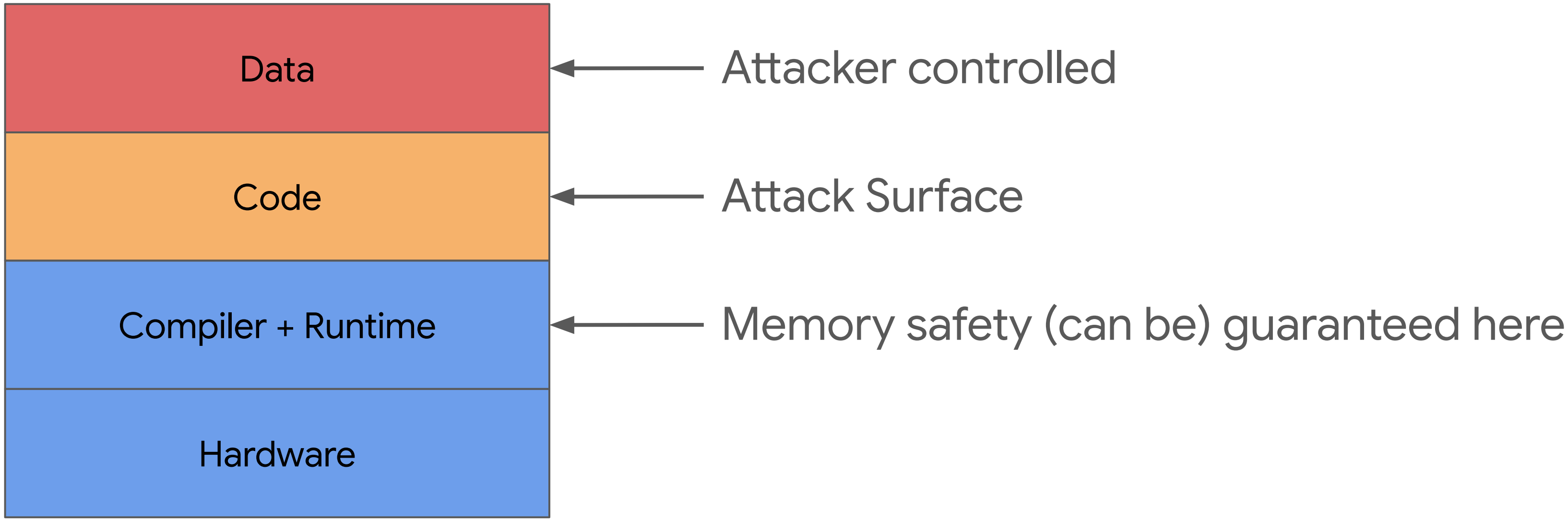
```
FUNC get(i, m):
  SPECULATE typeof(i) == Smi
  IF i < 0 RETURN
  IF i >= array.length RETURN
  M = ToNumber(m)
  __bounds_check__(i, array.length)
  A = array[i]
  RETURN M * A
```

Redundant bounds-check will be removed by compiler during code optimization...

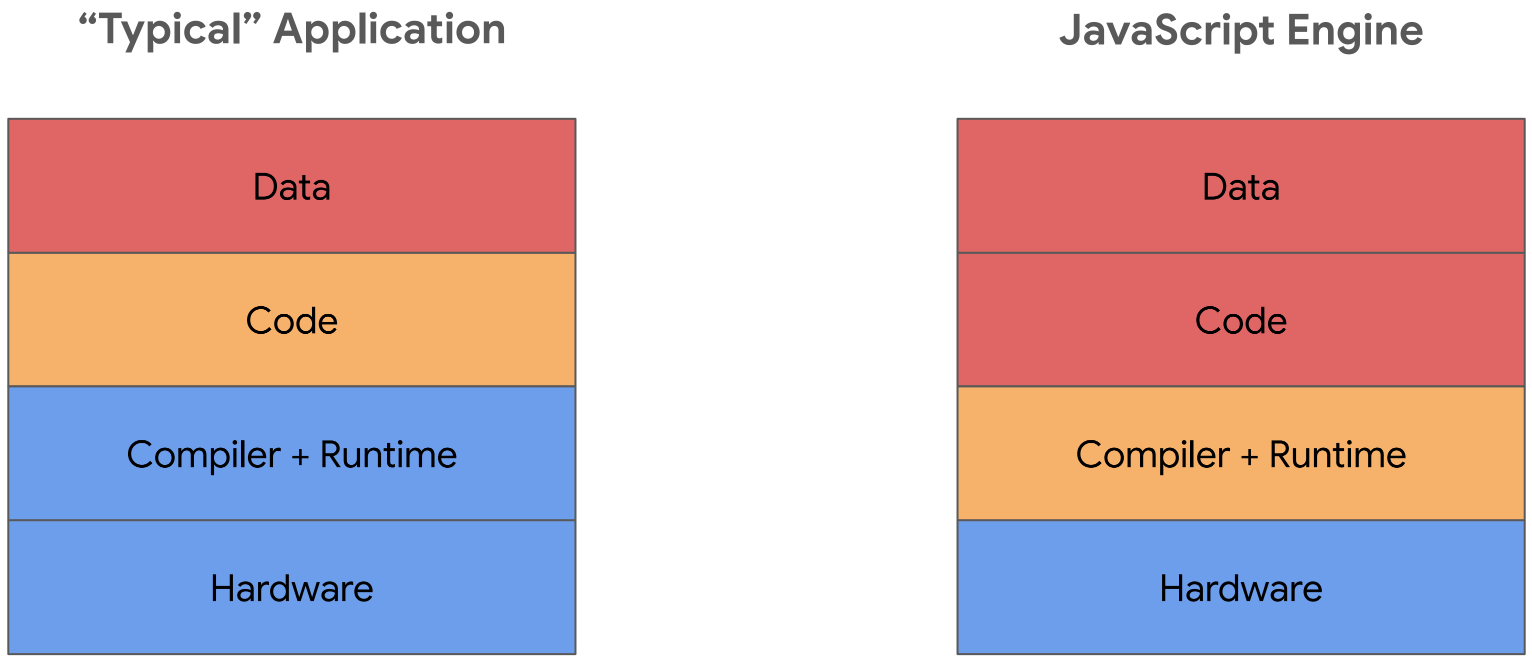
Except that it’s not redundant in this case 😖

Why JavaScript Engine Security is hard

“Typical” Application

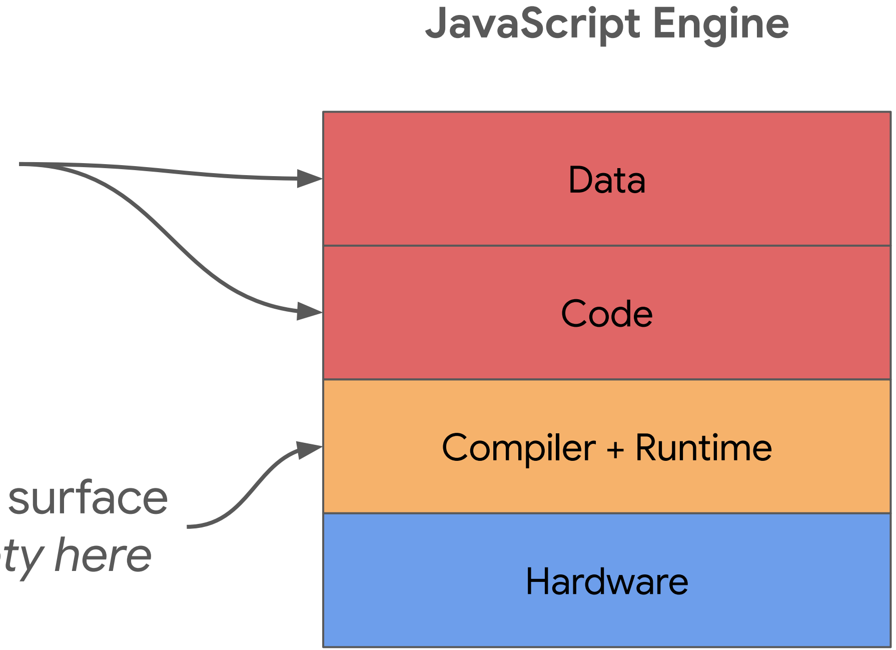


Why JavaScript Engine Security is hard



Why JavaScript Engine Security is hard

Attacker controlled

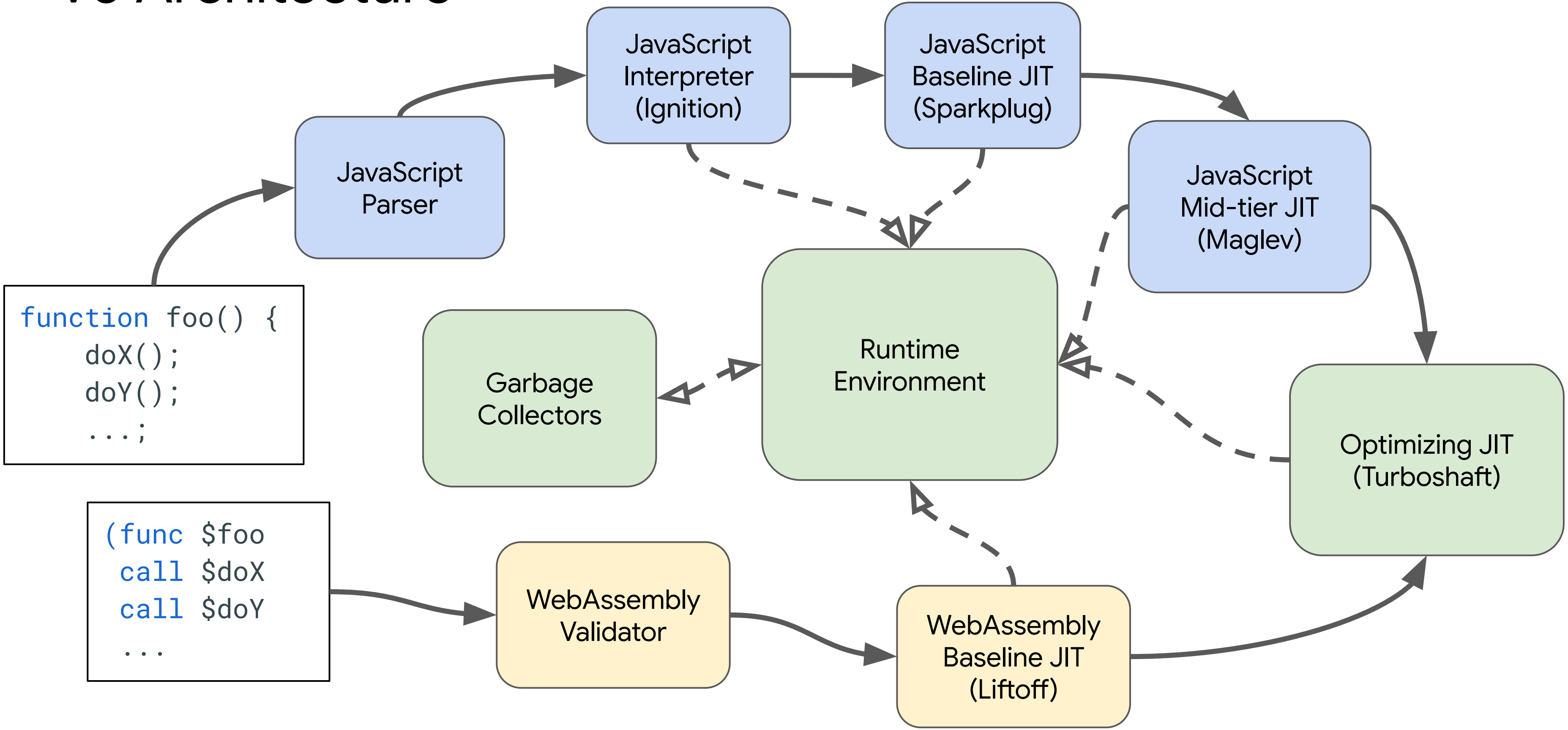


This is direct attack surface

=> Cannot guarantee memory safety here

Brief History of JS Engine Security

V8 Architecture

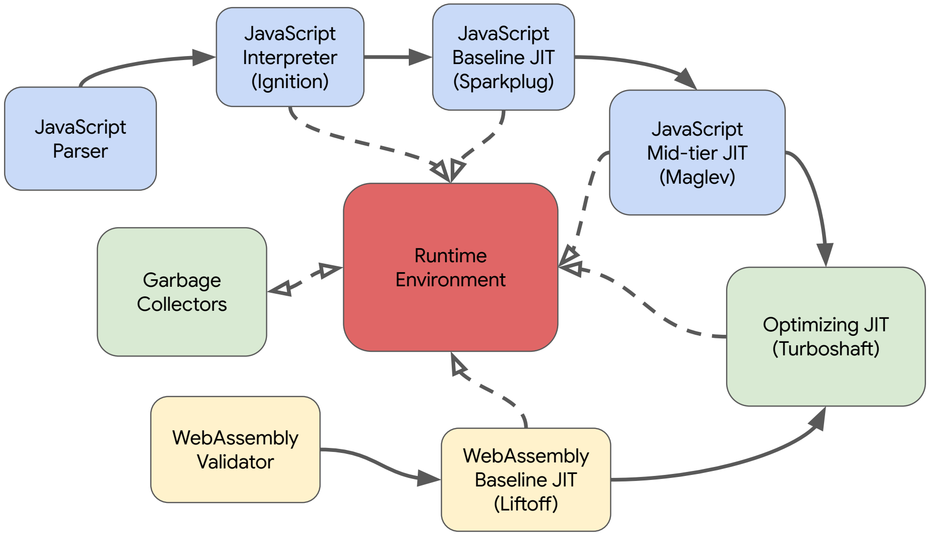


Brief History of JS Engine Bugs (Subjective)

- Phase 1 (<= ~2017): “Classic” runtime bugs
  - Bugs mostly in the runtime and builtin functions
  - Fairly local and somewhat shallow bugs
- Phase 2 (>= ~2017): Optimization bugs
  - Bugs deeper in the execution pipeline (e.g. JIT)
  - Some of the most complex bugs in software security?
- Phase 3 (>= ~2023): Wasm enters the picture
  - *Major* leap in complexity with WasmGC proposal
  - Brought complexity close to that of the JS pipeline
- Today: Mostly a mix of 2 and 3 (for the major engines)

Phase 1: Runtime Bugs

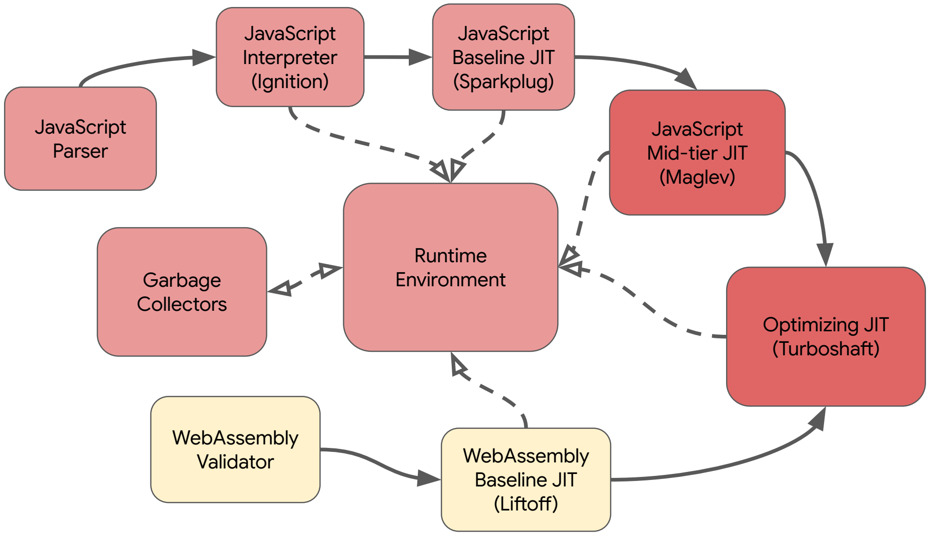
- By now “classic” JS engine bugs
- Typically bug pattern: unexpected callback in runtime functions which violates previous assumptions
    - But also: integer overflows, etc.
- Bugs often local to a single function
- Have *mostly* disappeared by now in the major JS engines




```
var a = [];
for (var i = 0; i < 100; i++) {
  a.push(i + 0.123);
}
var evil = {
  valueOf: function() {
    a.length = 0;
    return 10;
  }
};
var b = a.slice(0, evil);
```

Phase 2: Optimization Bugs

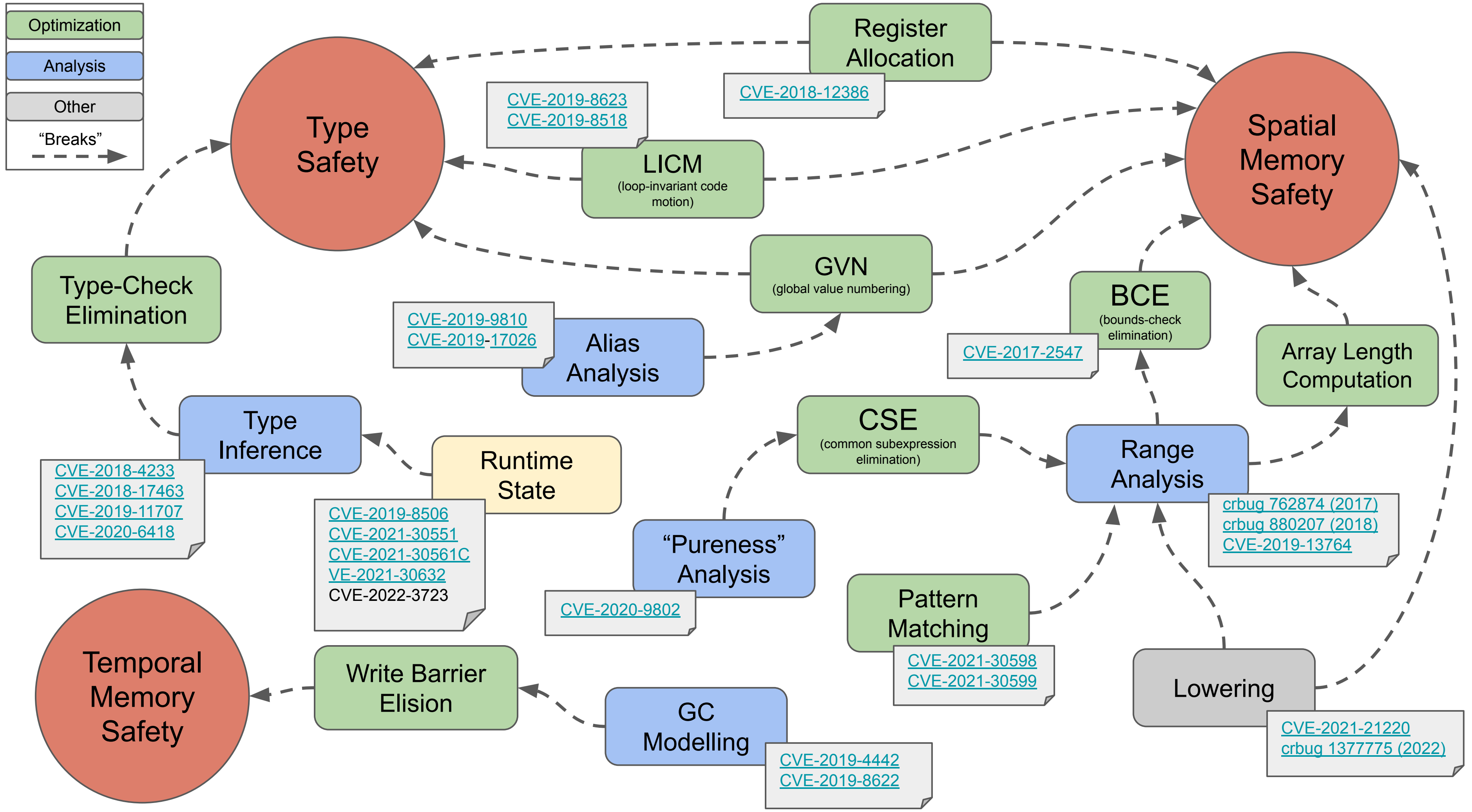
- Started with fairly simple (in retrospect!) JIT bugs
  - E.g. invalid bounds-check elimination
- Gradually became more and more complex, frequently involving multiple components of the engine
- Also more recently: bugs in the parser!



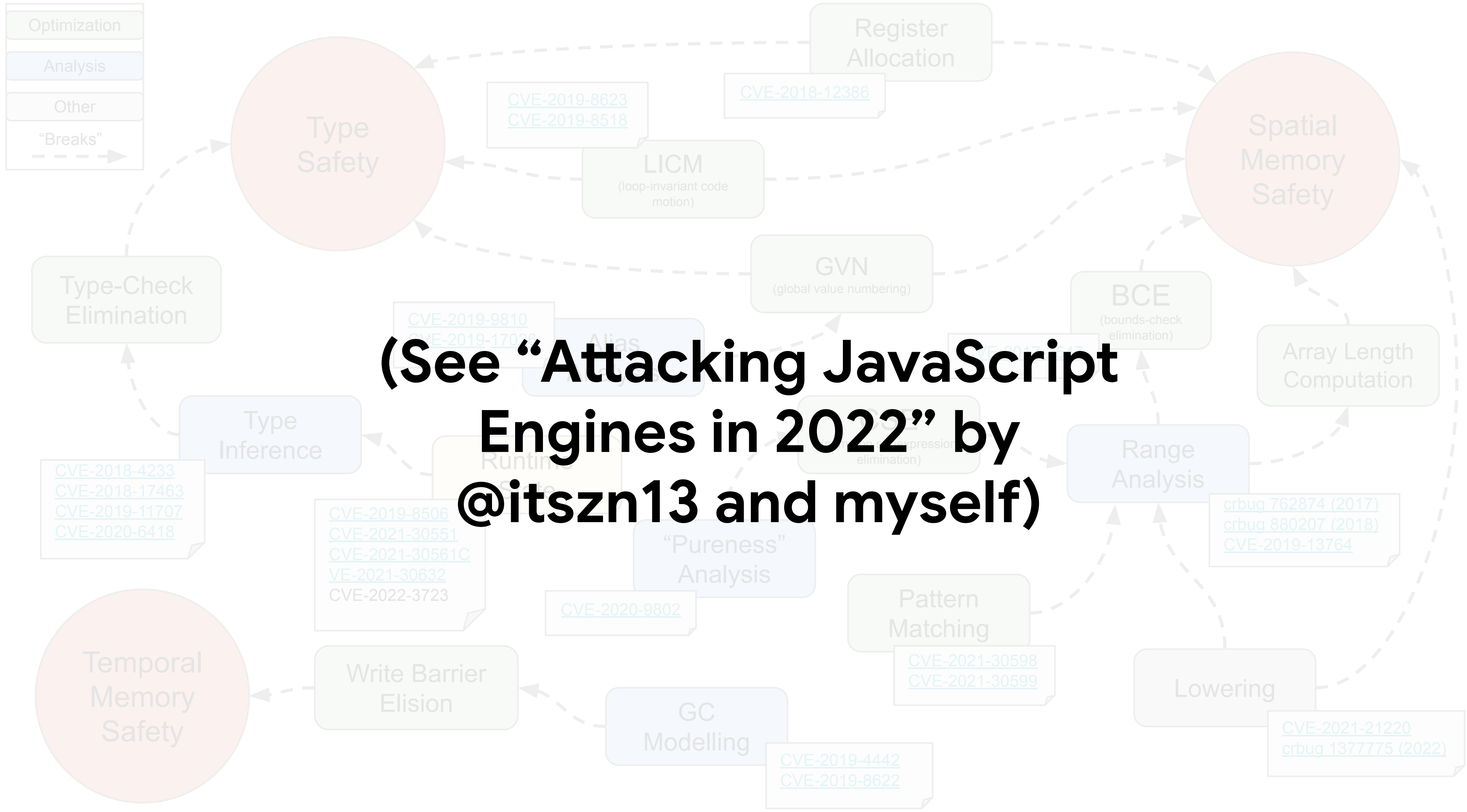
Exploiting the Math.expm1 typing bug in V8

02 Jan 2019

Minus zero behaves like zero, right?



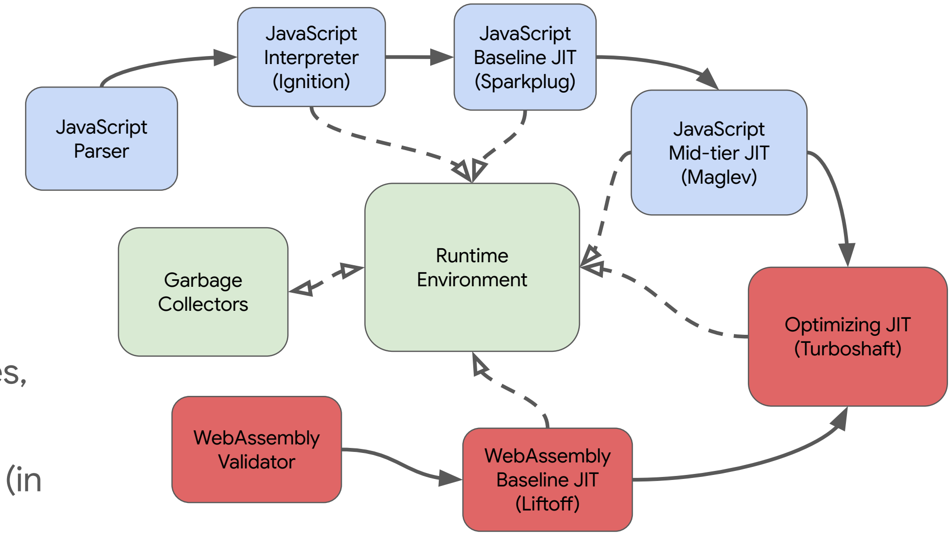
Optimization
Analysis
Other
"Breaks"
Type Safety
Spatial Memory Safety
Temporal Memory Safety
Register Allocation
CVE-2018-12386
CVE-2019-8623
CVE-2019-8518
LICM
(loop-invariant code motion)
GVN
(global value numbering)
Type-Check Elimination
CVE-2019-9810
CVE-2019-17026
Alias Analysis
BCE
(bounds-check elimination)
CVE-2017-2547
Array Length Computation
Type Inference
CVE-2018-4233
CVE-2018-17463
CVE-2019-11707
CVE-2020-6418
Runtime State
CVE-2019-8506
CVE-2021-30551
CVE-2021-30561C
VE-2021-30632
CVE-2022-3723
CSE
(common subexpression elimination)
Range Analysis
crbug 762874 (2017)
crbug 880207 (2018)
CVE-2019-13764
"Pureness" Analysis
CVE-2020-9802
Pattern Matching
CVE-2021-30598
CVE-2021-30599
Lowering
CVE-2021-21220
crbug 1377775 (2022)
Write Barrier Elision
GC Modelling
CVE-2019-4442
CVE-2019-8622



(See "Attacking JavaScript Engines in 2022" by @itszn13 and myself)

Phase 3: Wasm Bugs

- Wasm first generally available in ~2017
- BUT: that was "linear memory" Wasm:
  - Simple instruction set (basically loads+stores, arithmetic, logic operations, jumps, calls)
  - No object model, only linear memory range (in which memory corruption didn't matter)
  - Mostly only allowed compiling C/C++ to Wasm
- Then ~2023 WasmGC shipped
  - *Major* leap in complexity
  - Added powerful and highly complex object model + type system + garbage collection
  - ⇒ Major source of new vulnerabilities

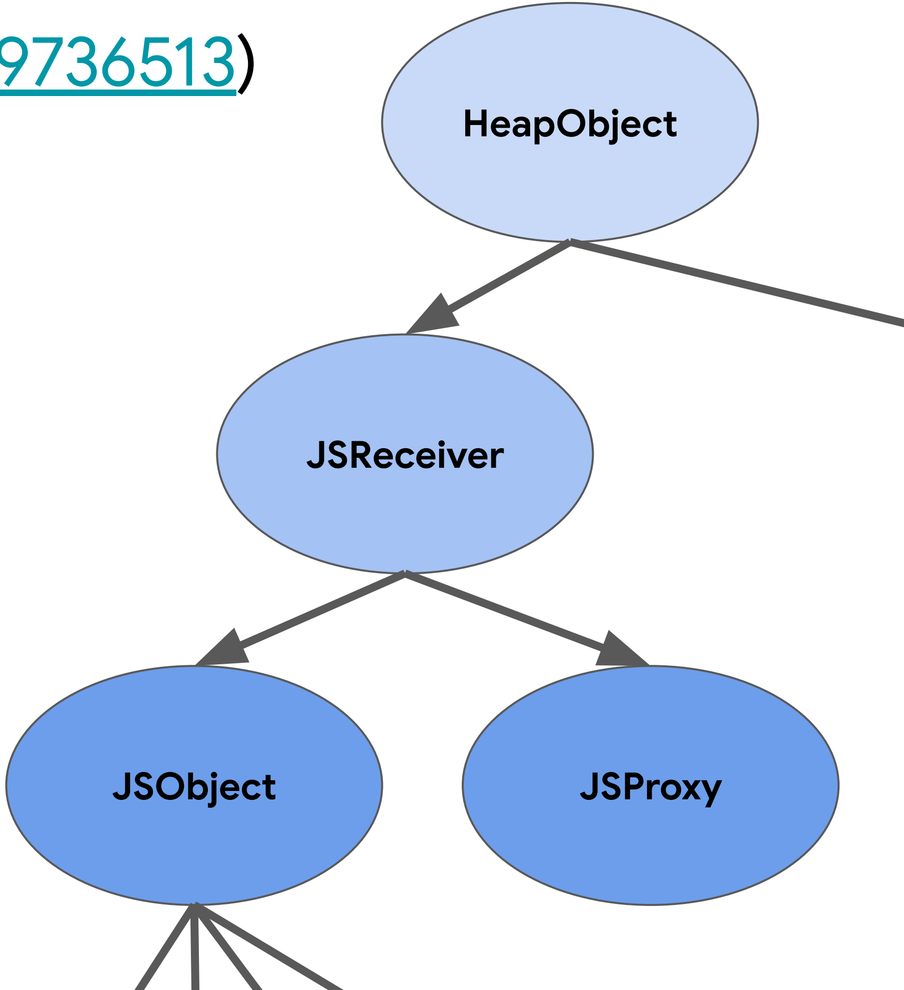


A new way to bring garbage collected programming languages efficiently to WebAssembly

Published 01 November 2023 · Tagged with WebAssembly

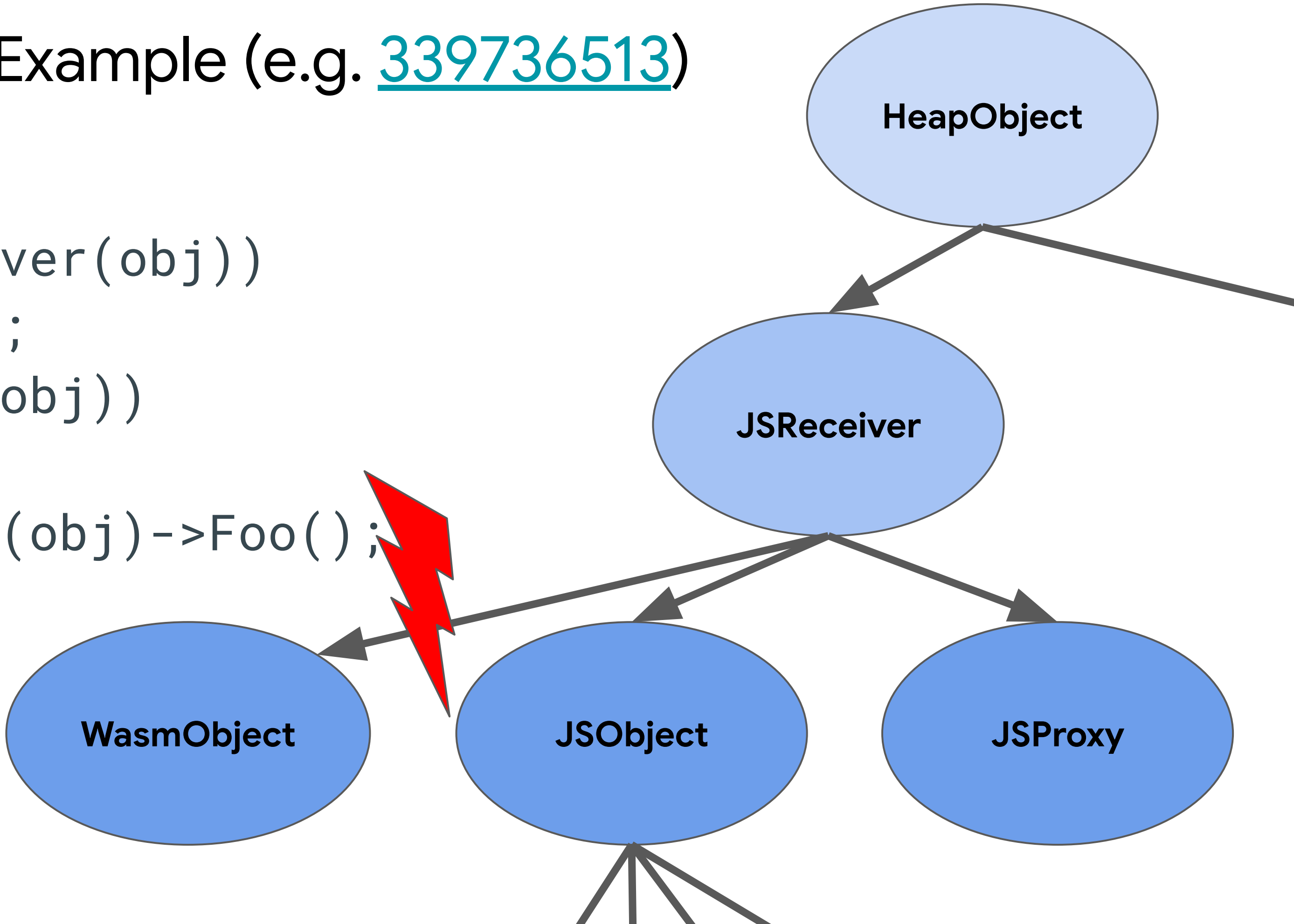
WasmGC Bug Example (e.g. 339736513)

```
// ...
if (!IsJSReceiver(obj))
  return false;
if (IsJSProxy(obj))
  return false
Cast<JSObject>(obj)->Foo();
```



WasmGC Bug Example (e.g. 339736513)

```
// ...
if (!IsJSReceiver(obj))
  return false;
if (IsJSProxy(obj))
  return false
Cast<JSObject>(obj)->Foo();
```



Defenses

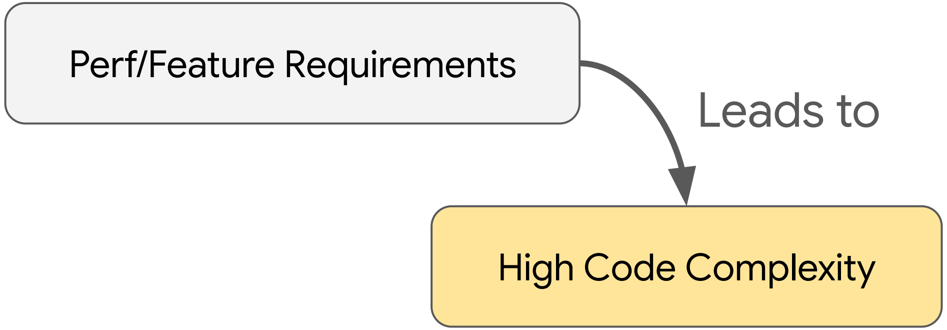
Where do vulnerabilities come from?

Where do vulnerabilities come from?

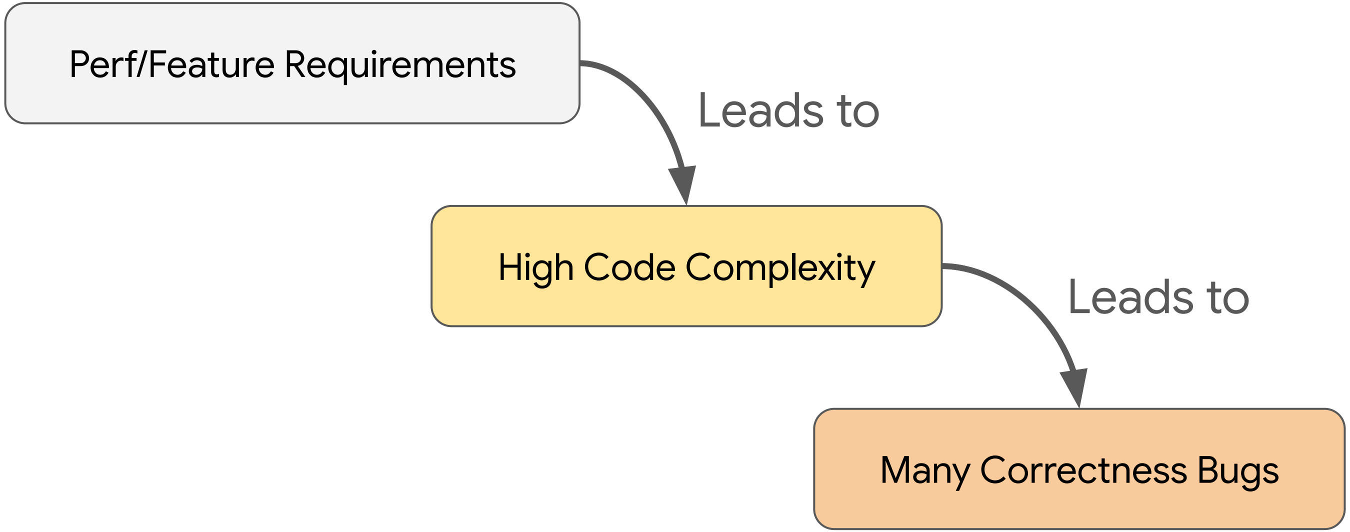
Perf/Feature Requirements

(in combination with highly complex and quickly evolving language spec)

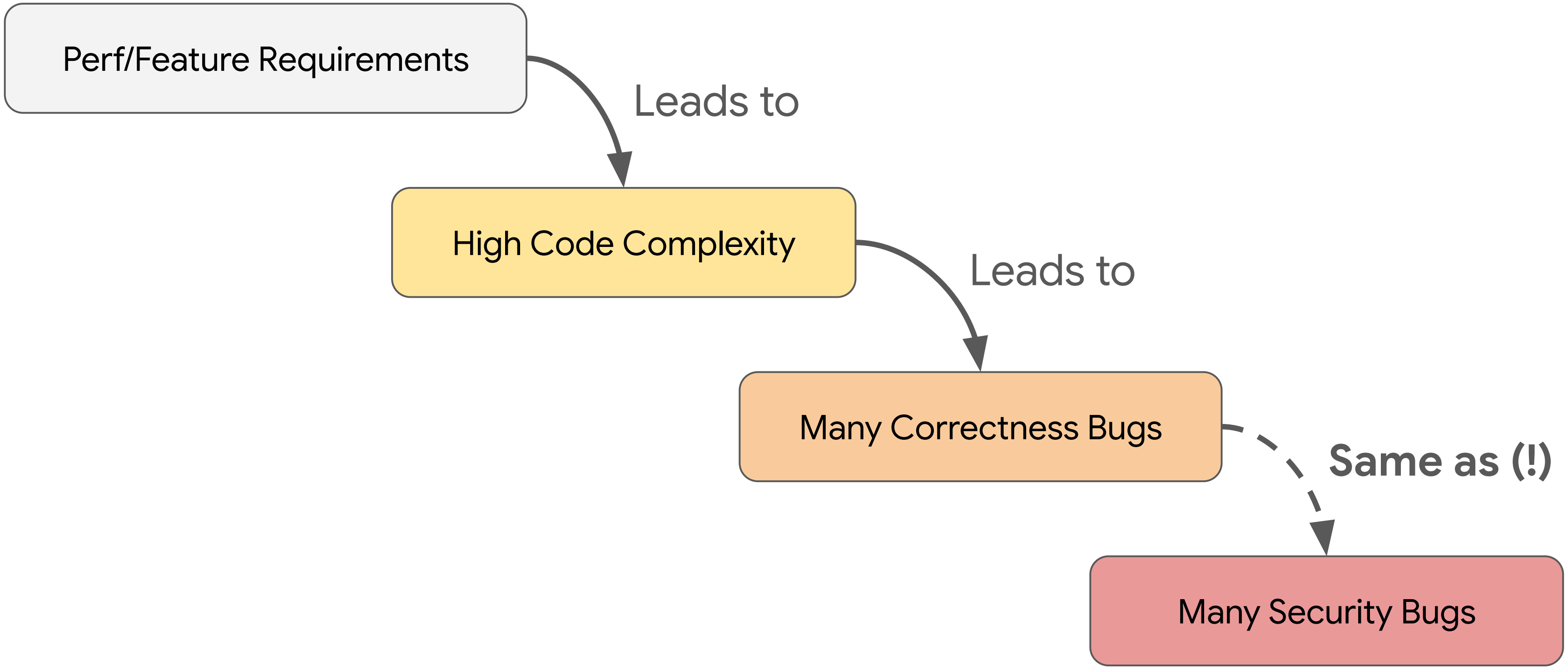
Where do vulnerabilities come from?



Where do vulnerabilities come from?

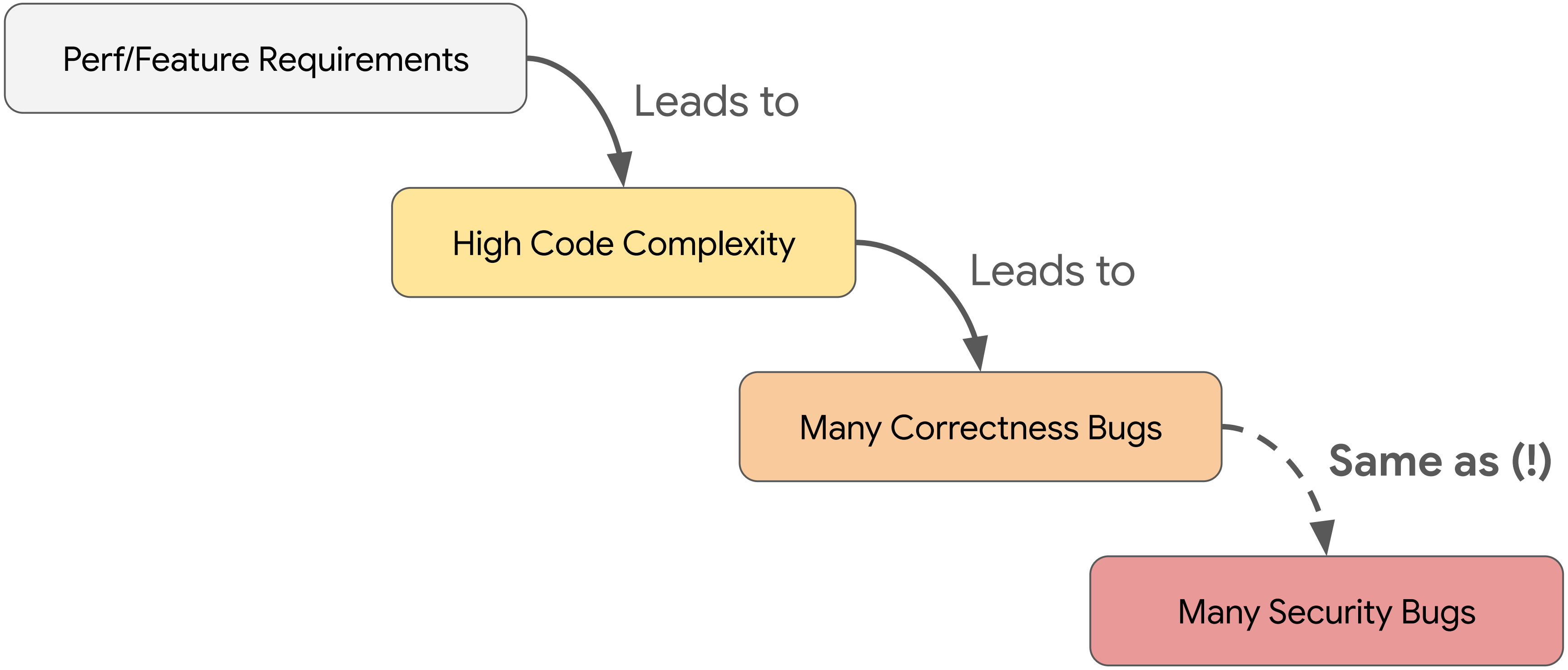


Where do vulnerabilities come from?

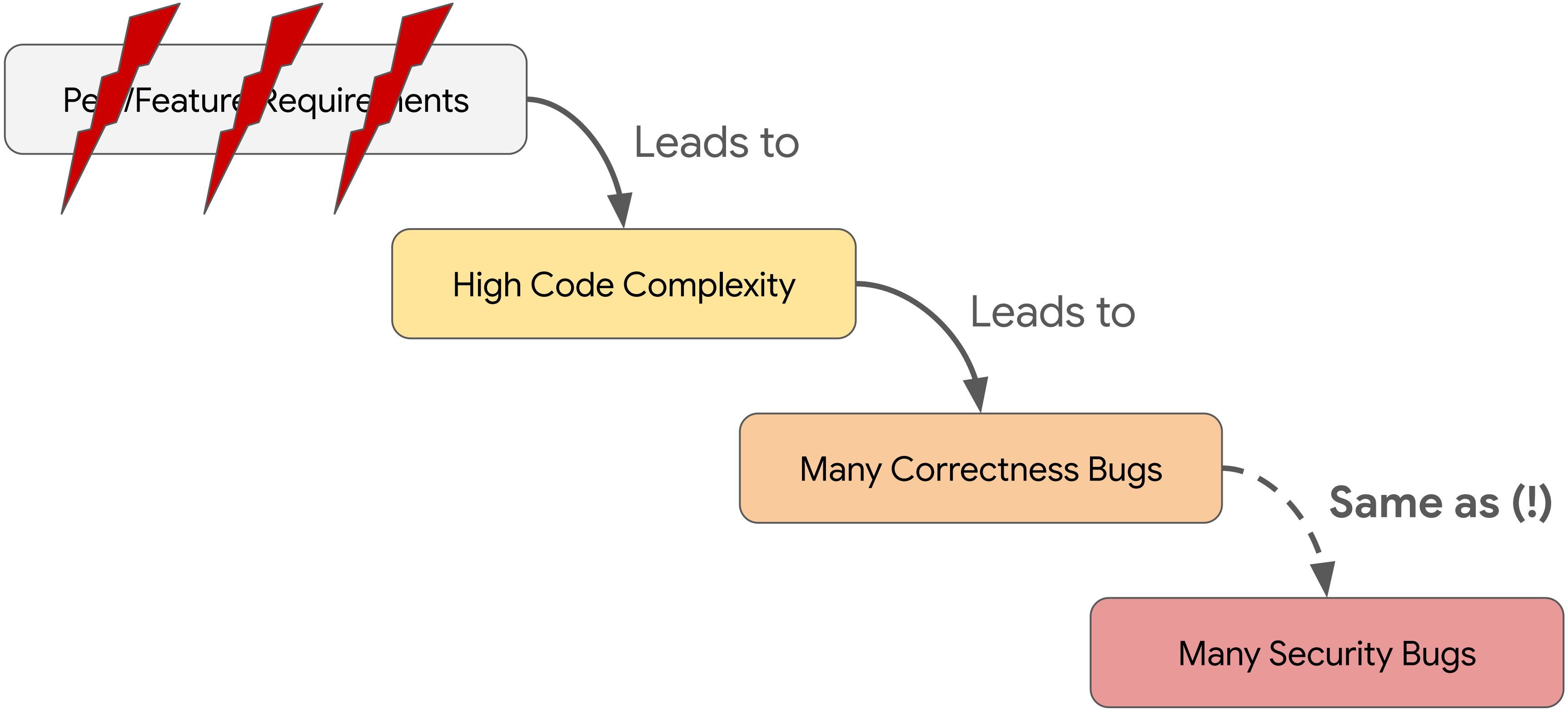


What to do?

What to do?

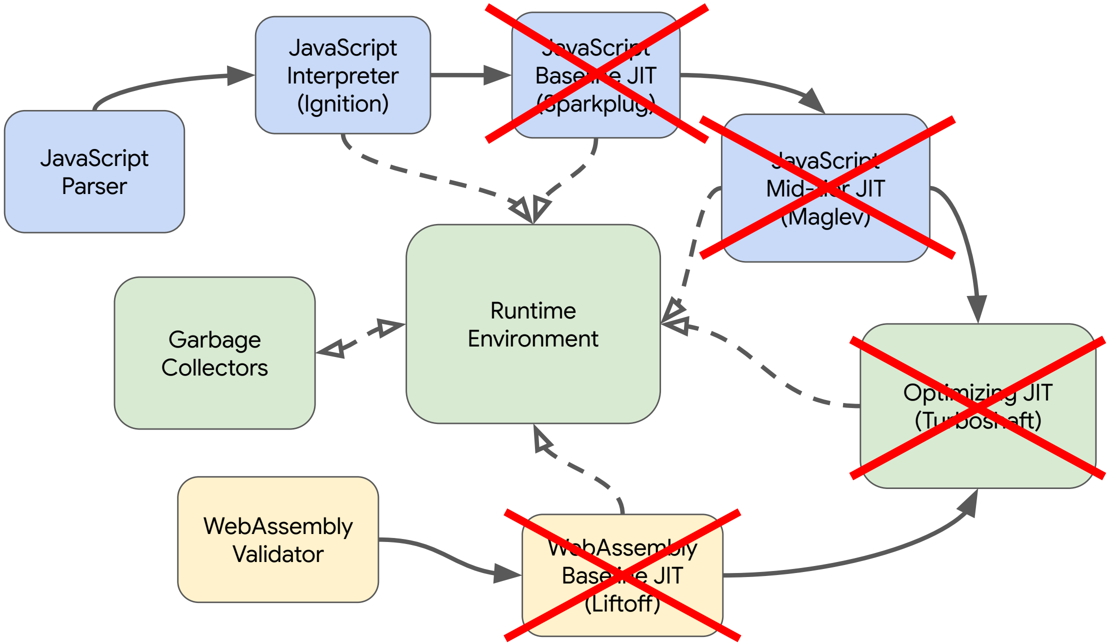

Perf/Feature Requirements
Leads to
High Code Complexity
Leads to
Many Correctness Bugs
Same as (!)
Many Security Bugs

What to do?

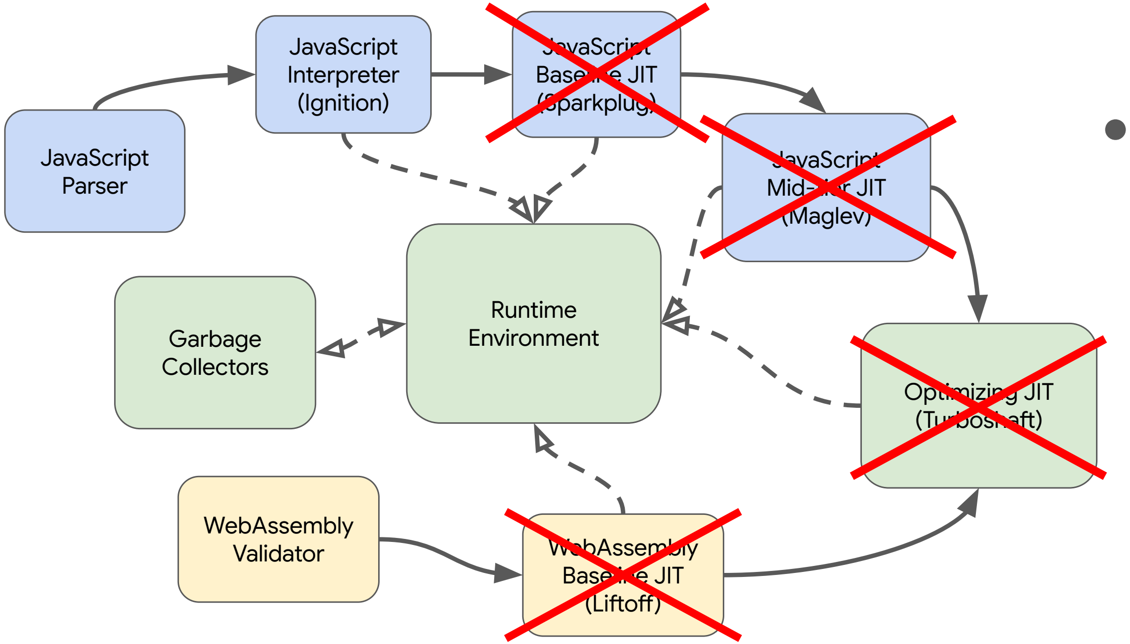

Pe /Featur Require ents
Leads to
High Code Complexity
Leads to
Many Correctness Bugs
Same as (!)
Many Security Bugs

Going Jitless?

- Idea: disable compilers
- Obvious attack surface reduction (always good!)
- In addition: may allow enabling additional mitigations (mostly W^X)
- But: potentially dramatic performance penalty (anything from **5%-90%**)



Going Jitless? What about Wasm (in V8)?



- Problem: Wasm doesn't need/use an interpreter
- Options:
  - Add a Wasm interpreter

    ⇒ new attack surface :/

    (unless written in memory-safe language?)

NAN WANG (SAKURA) & ZILING CHEN (R1ND0)

ENHANCED INSECURITY MODE: 23 RCES IN EDGE'S "SAFE" WEBASSEMBLY INTERPRETER

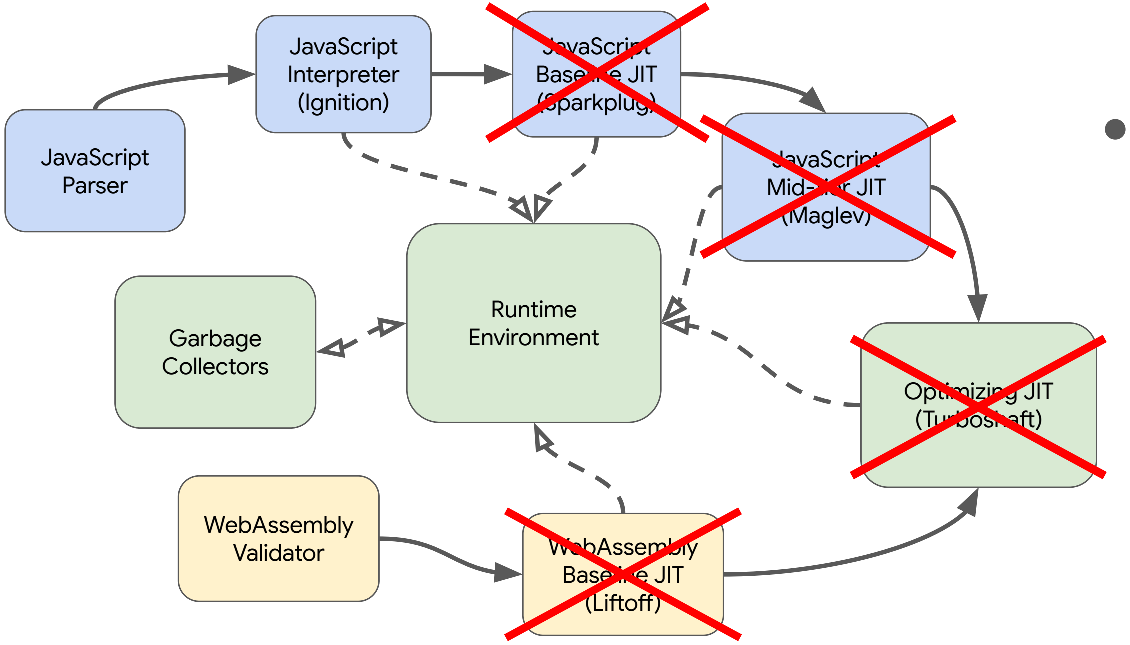
Abstract

Microsoft Edge's Enhanced Security Mode was designed as the ultimate defense when browsing unfamiliar websites. By disabling JIT compilation and forcing all WebAssembly code through an interpreter called DrumBrake, Microsoft promised protection against the most common classes of browser exploitation. The irony? This security feature became a massive attack surface itself.

Over the past year of continuous research, we discovered and reported 23 remote code execution vulnerabilities in DrumBrake, completely defeating every security guarantee that Enhanced Security Mode promises. Our findings span multiple vulnerability classes: type confusion bugs arising from incorrect WebAssembly GC reference handling, out-of-bounds memory access due to flawed reference stack index calculations, use-after-free conditions caused by missing garbage collection write barriers, and critical control flow errors where the interpreter simply forgot to stop executing.

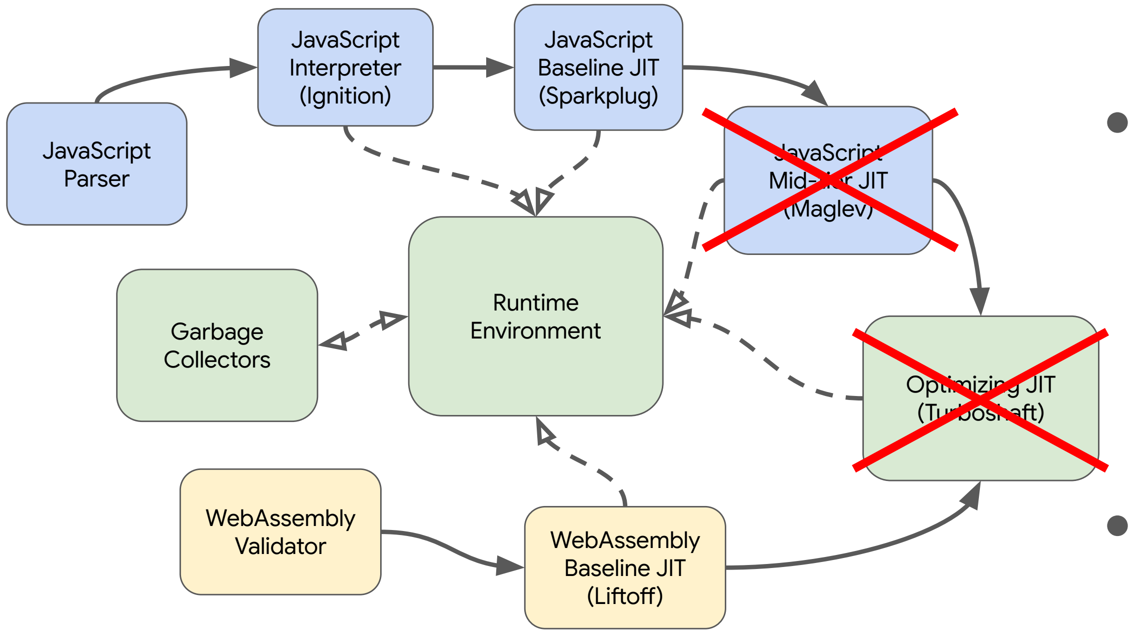
This talk provides a comprehensive technical deep-dive into DrumBrake's architecture, our systematic vulnerability hunting methodology, and complete exploitation chains that transform subtle interpreter bugs into arbitrary code execution. We will demonstrate how a single overlooked function call, how a duplicated index calculation, and how an incorrect calling sequence each independently led to full browser compromise. All 23 vulnerabilities have been patched by Microsoft, and this research earned recognition in Microsoft's security researcher acknowledgements for nearly a year straight.

Going Jitless? What about Wasm (in V8)?



- Problem: Wasm doesn’t need/use an interpreter
- Options:
  - Add a Wasm interpreter
    ⇒ new attack surface : /
    (unless written in memory-safe language?)
  - Disable Wasm : /

Going Jitless? What about Wasm (in V8)?



- Problem: Wasm doesn't need/use an interpreter
- Options:
  - Add a Wasm interpreter
    ⇒ new attack surface :/
    (unless written in memory-safe language?)
  - Disable Wasm :/
  - Keep Baseline Wasm JIT :|
- Hypothesis: just disabling optimizing JITs is sufficient

Going Jitless? The Data!

- Tracking sheet for V8 bugs that are known to be exploitable
- Sources: ITW, exploit competitions, V8CTF
- Seems to confirm:
  - JITs account for **~50%** of exploitable vulnerabilities
  - Baseline JITs are rarely the source of vulnerabilities

#	Issue	First Exploited	Description	Exploit requires V8 Sandbox Bypass	Exploit requires optimizing JITs (Turbofan & Maglev)	Exploit requires any JITs (Liftoff, Sparkplug, Maglev & Turbofan)	Variant	JavaScript or WebAssembly	Introduced by	Introduced in
1	386565144	V8CTF	Incorrect optimization in Maglev	Yes	Yes	Yes	No	JavaScript	Performance Work	2024
2	391907159	V8CTF	Wasm JIT allocation UaF	No*	Yes	Yes	No	WebAssembly	Performance Work	2024
3	398065918	V8CTF	Improper allocation folding in Maglev	Yes	Yes	Yes	No	JavaScript	Performance Work	2024
4	400052777	V8CTF	Incorrect handling of aliases during ElementsKind transitions	Yes	Yes	Yes	No	JavaScript	Performance Work	2024
5	403364367	V8CTF	Invalid handling of Wasm stack frames during stack walking	Yes	No	No	No	Both	Feature Work	2020
6	420636529	ITW	Logic error in store-store elimination	Yes	Yes	Yes	No	JavaScript	Performance Work	2024
7	422313191	TyphoonPWN	Invalid WebAssembly type canonicalization	Yes	No	No	Yes	WebAssembly	Performance Work	2025
8	427663123	ITW	Invalid hole check elision in interpreter	Yes	Probably	Probably	No	JavaScript	Performance Work	2023
9	433533359	V8CTF	Concurrent modification of Wasm code	No	No	No	No	N/A	Feature Work	2018 (?)
10	430344952	V8CTF	Divergence between preparser and parser	Yes	No	No	No	JavaScript	Performance Work	2016 (?)
11	436181695	V8CTF	Invalid parsing of 'await using' in c-style loops	No	No	No	No	JavaScript	Feature Work	2025
12	445380761	ITW	Invalid integer optimization on Arm64	Yes	Yes	Yes	No	JavaScript	Performance Work	2016

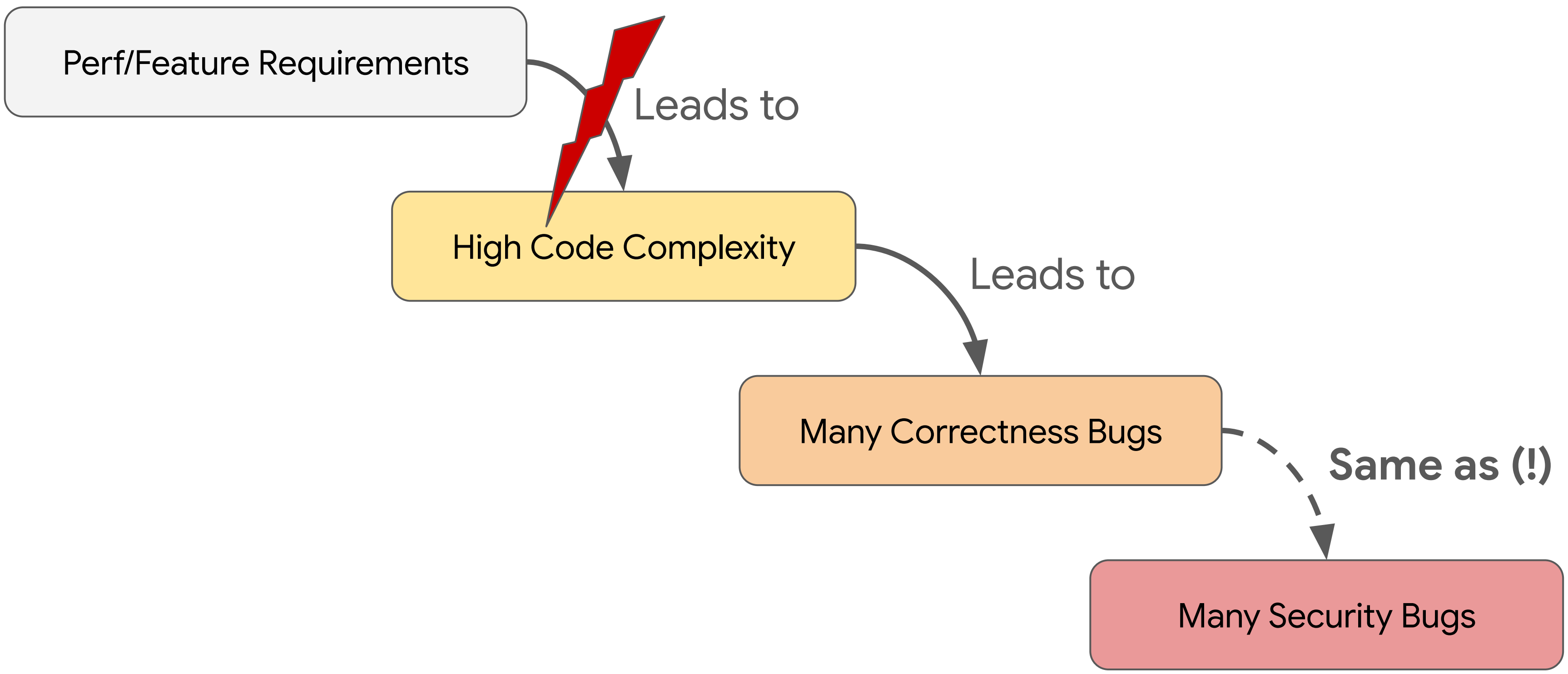
Going Jitless?

Worth emphasizing:

50% fewer bugs != 50% fewer attacks

⇒ Jitless isn’t the solution (but can still be a useful tool!)

What to do?


Perf/Feature Requirements
Leads to
High Code Complexity
Leads to
Many Correctness Bugs
Same as (!)
Many Security Bugs

Seems difficult?

Home | Blog | Docs | Tools | JS/Wasm features | Research

Land ahoy: leaving the Sea of Nodes

Published 25 March 2025 · Tagged with JavaScript internals

V8's end-tier optimizing compiler, Turbofan, is famously one of the few large-scale production compilers to use Sea of Nodes (SoN). However, since almost 3 years ago, we've started to get rid of Sea of Nodes and fall back to a more traditional Control-Flow Graph (CFG) Intermediate Representation (IR), which we

Some Examples

Sea of Nodes: elegant but impractical for JavaScript

So, to recapitulate, here are the main problems we have with Sea of Nodes and Turbofan:

1. It's **too complex**. Effect and control chains are hard to understand, leading to many subtle bugs. Graphs are hard to read and analyze, making new optimizations hard to implement and refine.

Saying goodbye to asm.js

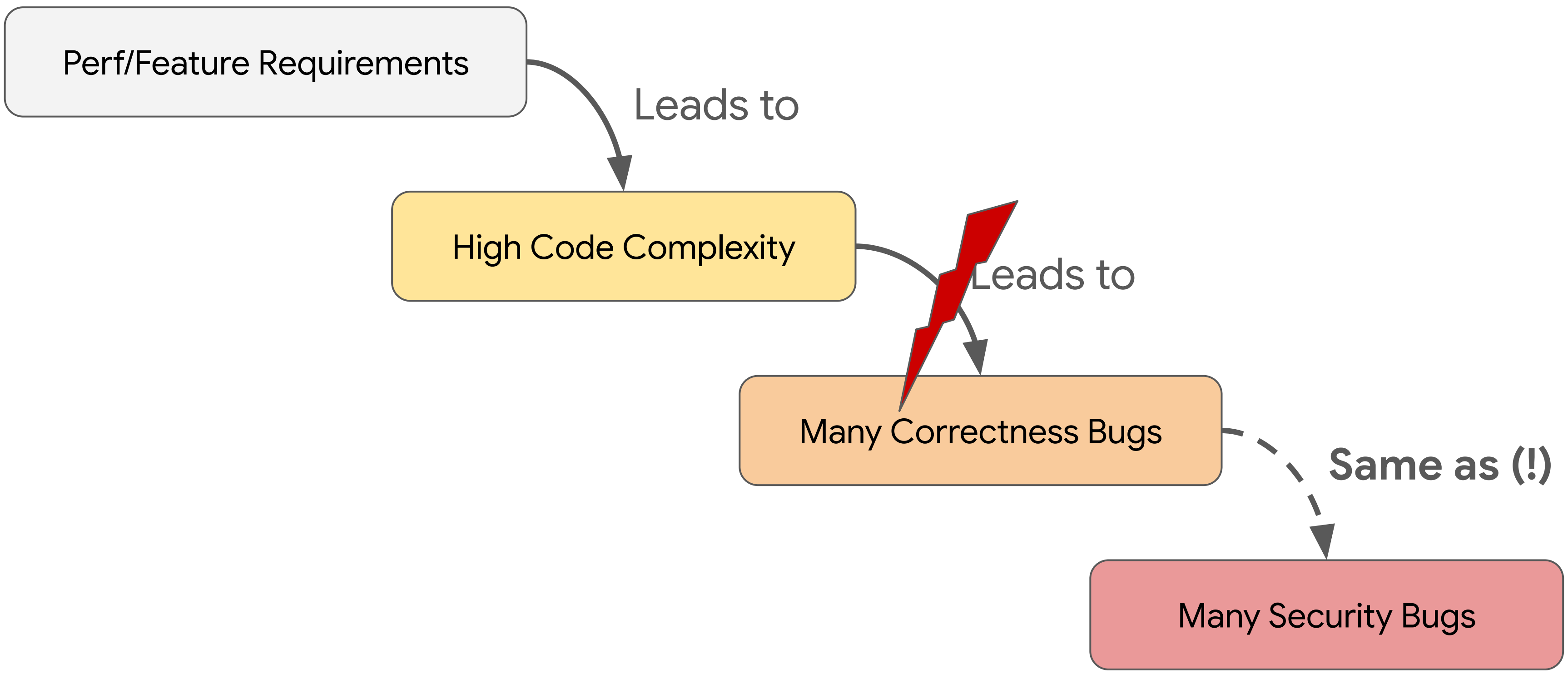
May 20, 2026 • Ryan Hunt

> Axe-time, sword-time, shields are sundered,
> Wind-time, wolf-time, ere the world falls.
> – *Völuspá*, Poetic Edda

As of Firefox 148, SpiderMonkey's asm.js optimizations are disabled by default, and we plan to remove the code entirely in a future release.

If you maintain a site that uses asm.js, nothing will break. asm.js is just a subset of plain JavaScript, so the code keeps running through our regular JIT just like any other script. That said, recompiling to WebAssembly will get you faster execution and smaller binaries.

What to do?



Perf/Feature Requirements
Leads to
High Code Complexity
Leads to
Many Correctness Bugs
Same as (!)
Many Security Bugs

Avoiding Bugs 🐛

- In theory, it's possible to write correct compilers... :)
- In practice, bugs happen
- Good testing, fuzzing, auditing etc. are crucial
  - More on that later!
- Also: good software design can prevent many bugs (or make them shallow)!
  - Example: JSC's `doesGC` verification made these bugs near-trivial to discover
  - Many similar verifiers elsewhere
- ... but it won't find/prevent every bug

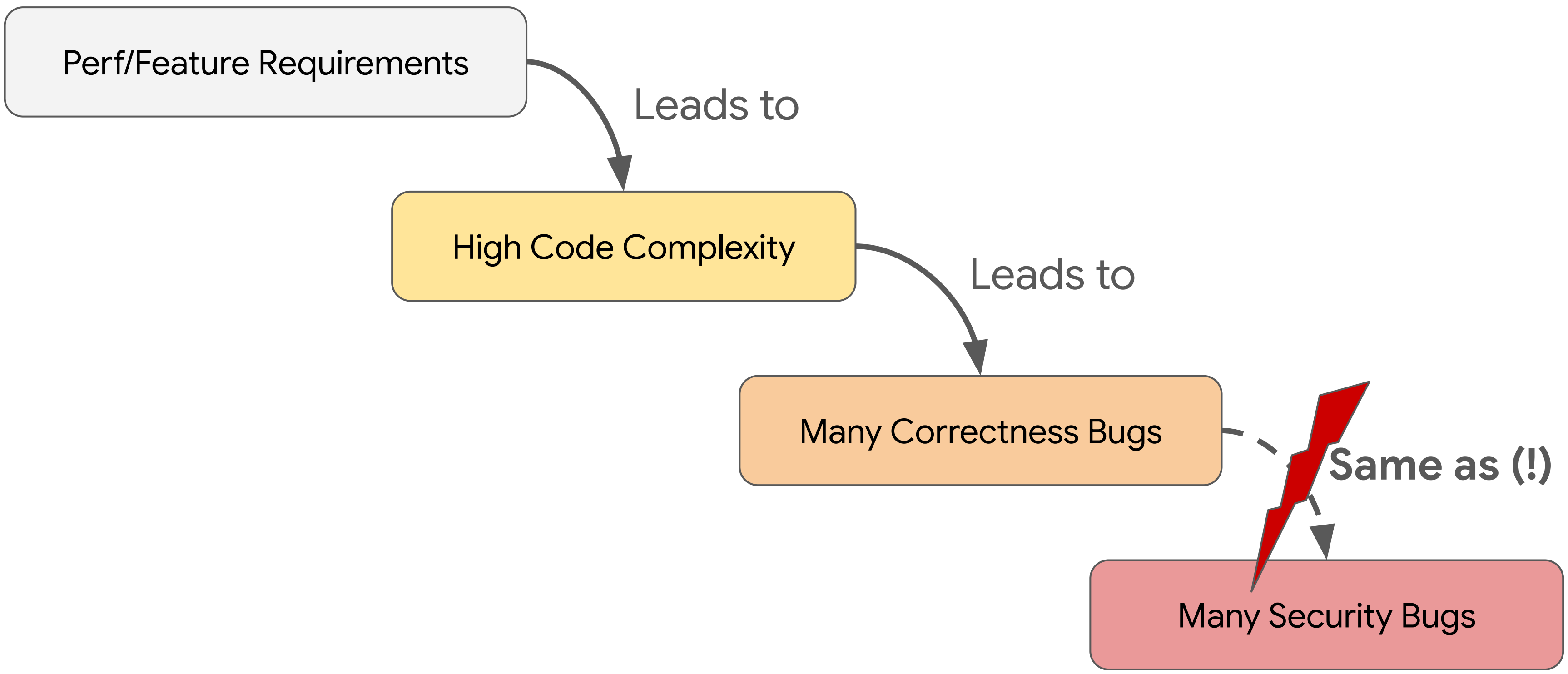
Not a security strategy...

Write Bug Free Code

In Complex Code Base

Big, hard problem

What to do?

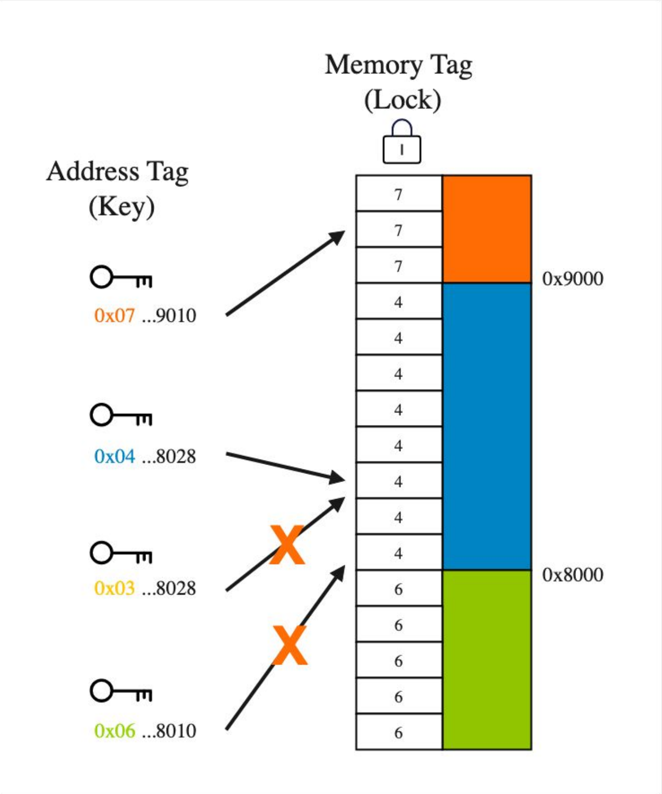

Perf/Feature Requirements
Leads to
High Code Complexity
Leads to
Many Correctness Bugs
Same as (!)
Many Security Bugs

Goal: decouple correctness from security!

Memory Tagging Extension (MTE)

Memory Tagging Extension (MTE)

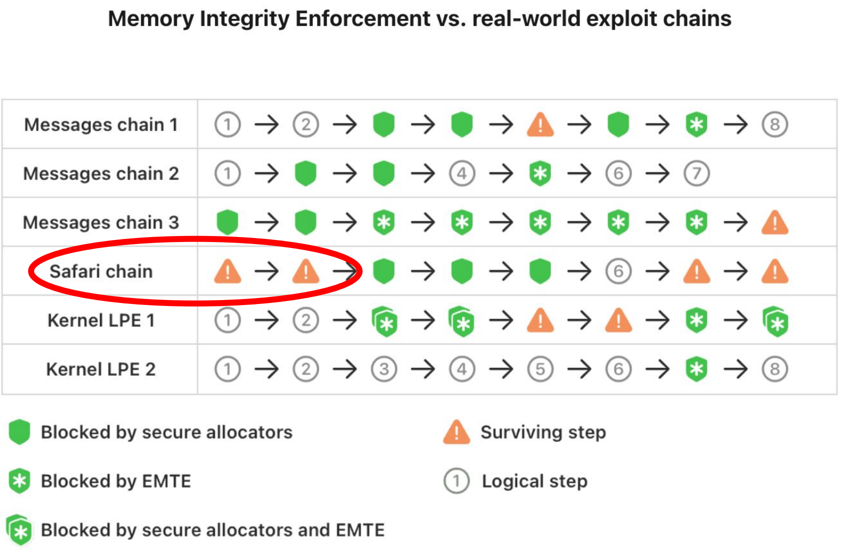
- Hardware feature for memory safety enforcement
- Basic idea: add tag to pointers and memory, enforce that they match
- Can mitigate a number of bug types, e.g.
  - Linear OOB accesses
  - Use-after-Free
- See ARM's documentation, Project Zero's analysis, or Apple's blog post for details



https://developer.arm.com/documentation/108035/0100/How-does-MTE-work-

Memory Tagging Extension (MTE)

- **Unlikely to have much impact in JavaScript engines**
  - (But most likely elsewhere!)
- Typical bugs there are too powerful:
  - Arbitrary OOB reads+writes
  - Arbitrary type confusions
- Also, custom pointer encodings leave no space for MTE tags...
- Apple seems to have come to a similar conclusion



https://security.apple.com/blog/memory-integrity-enforcement/

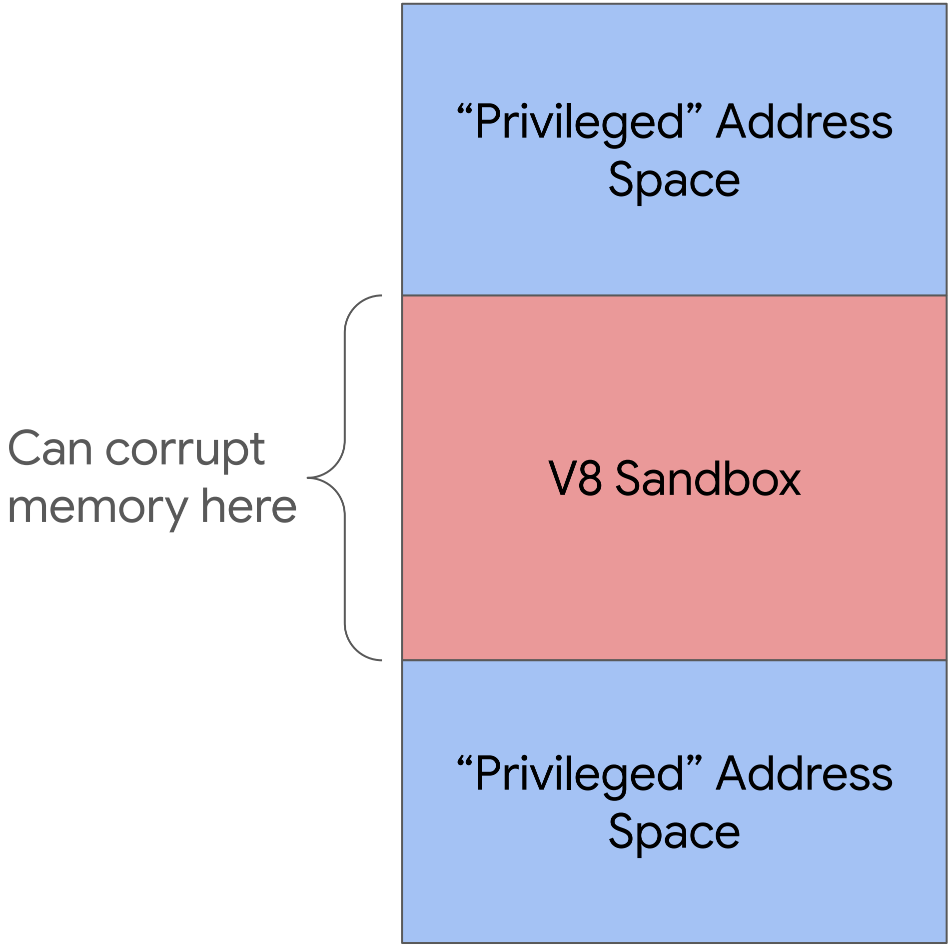
Sandboxing

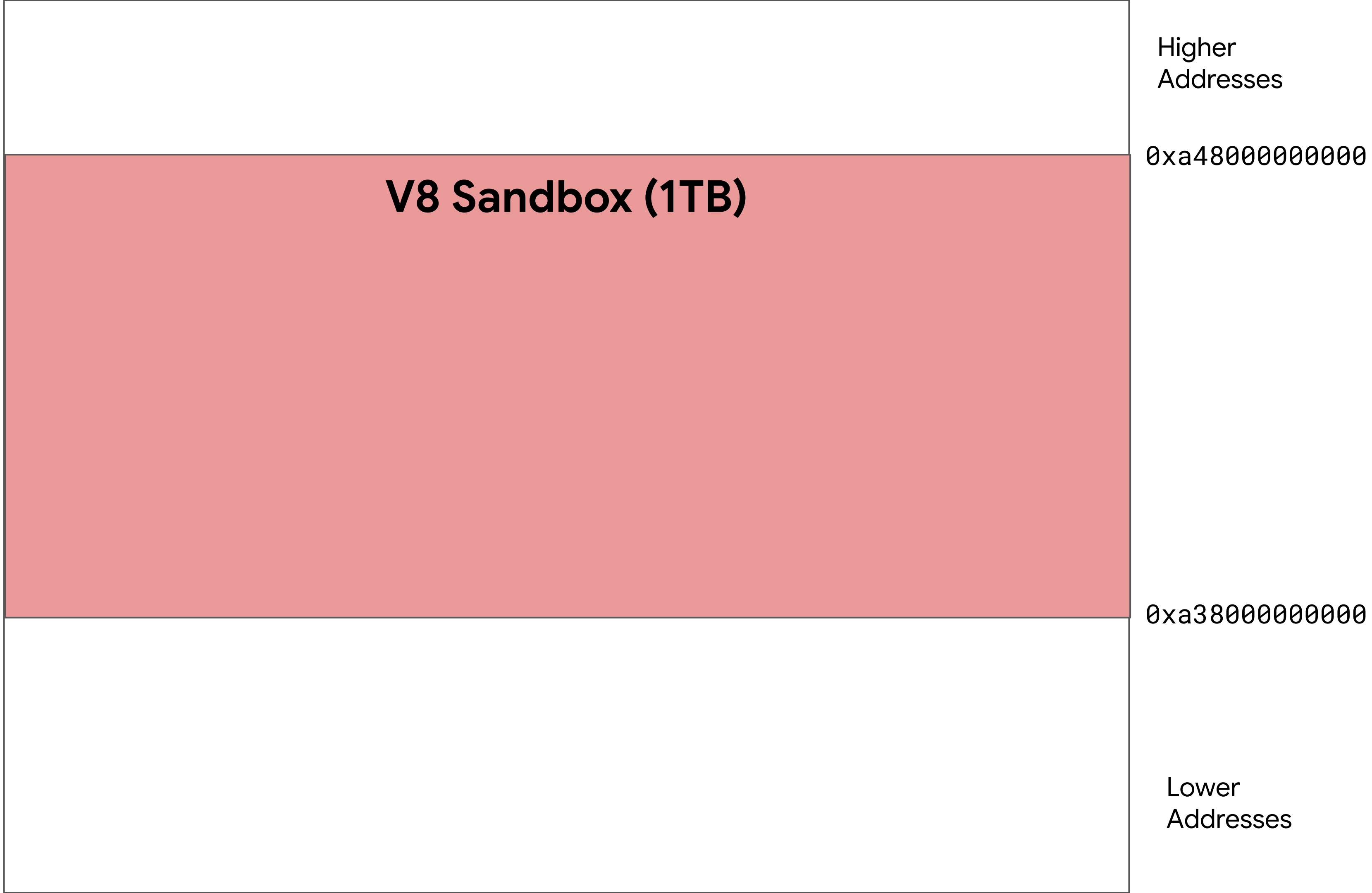
A different approach...

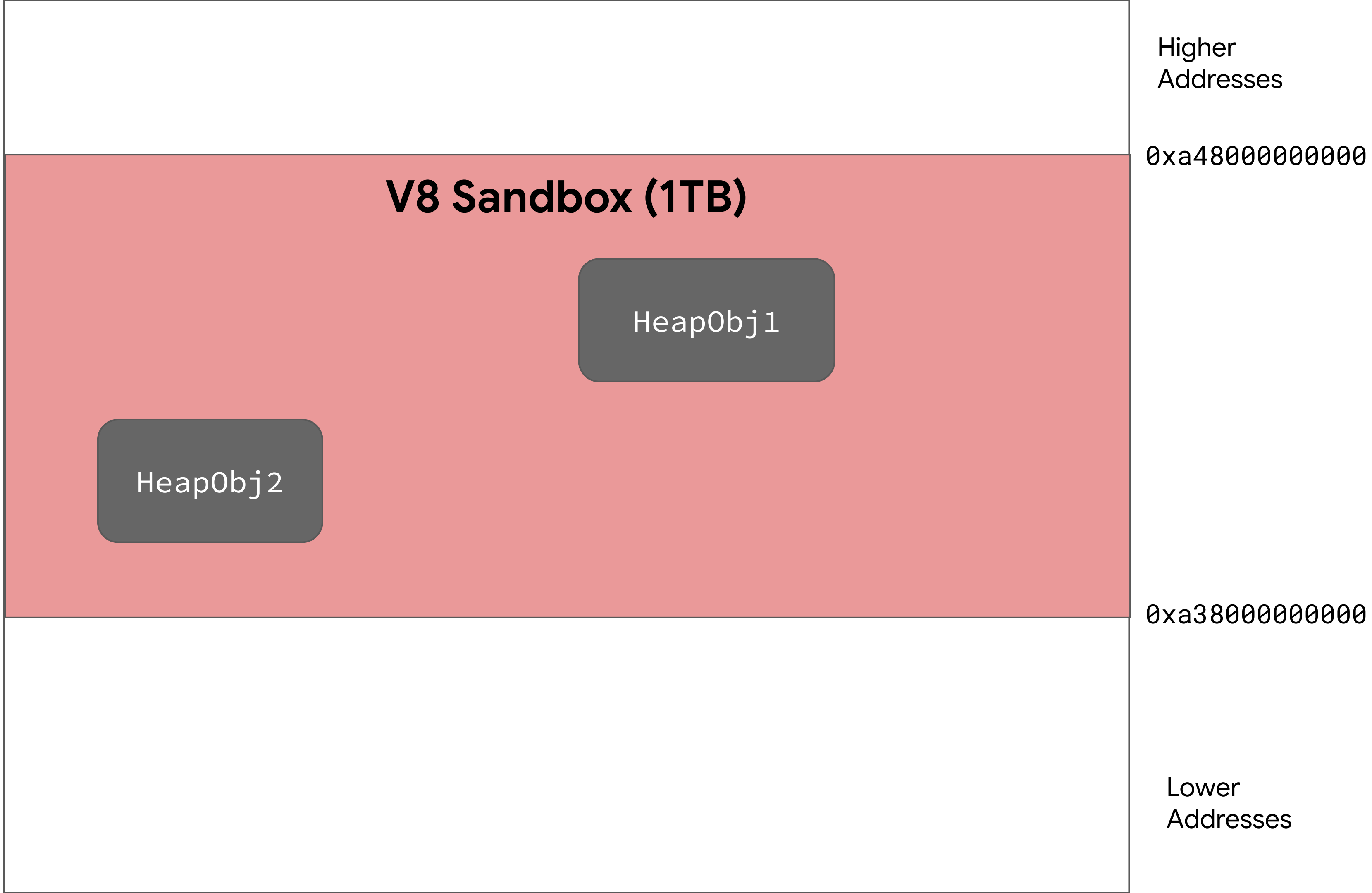
Idea:

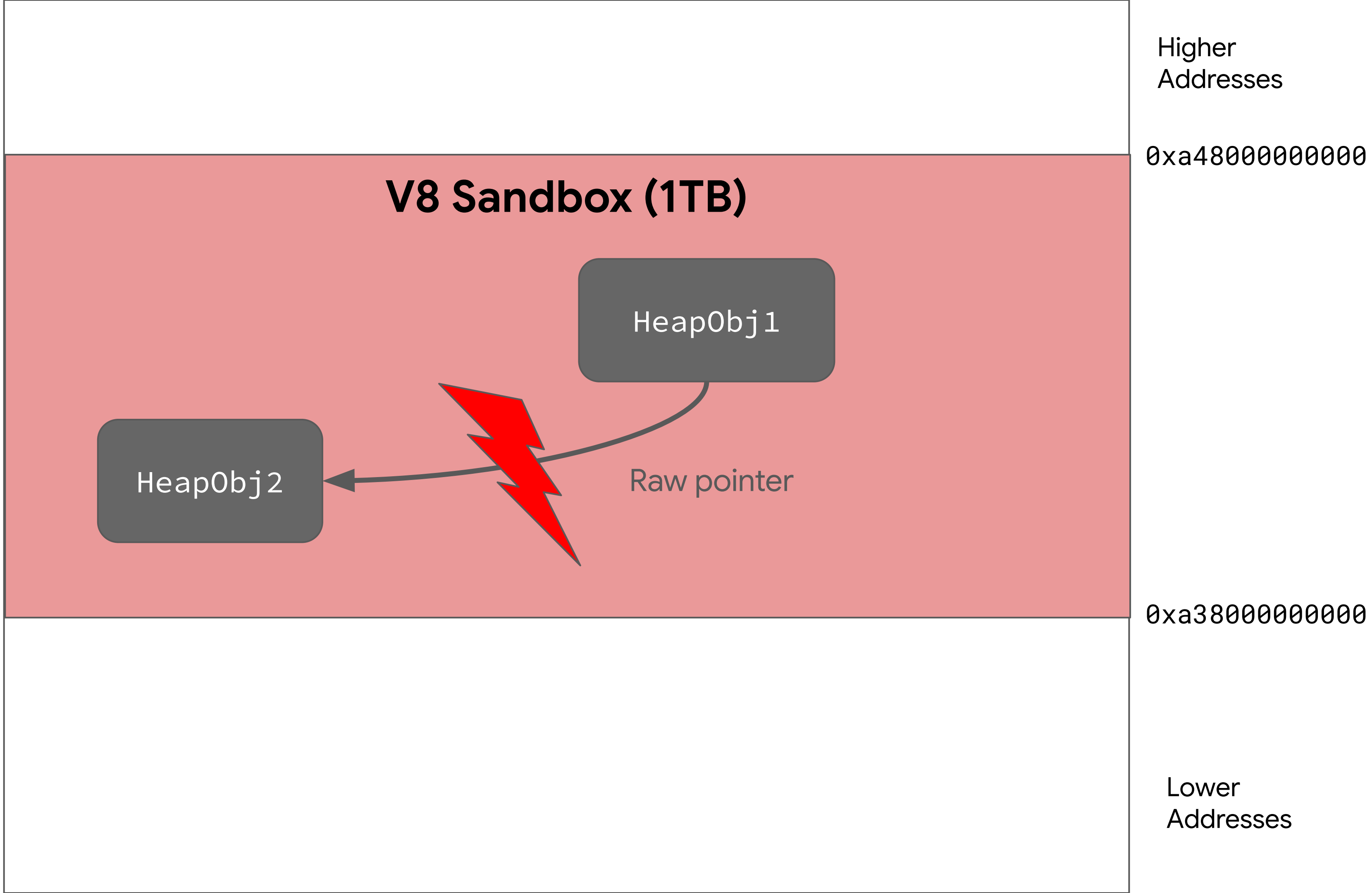
- Accept that bugs will happen and that memory will be corrupted
- Limit which memory can be corrupted
- Make that a security boundary

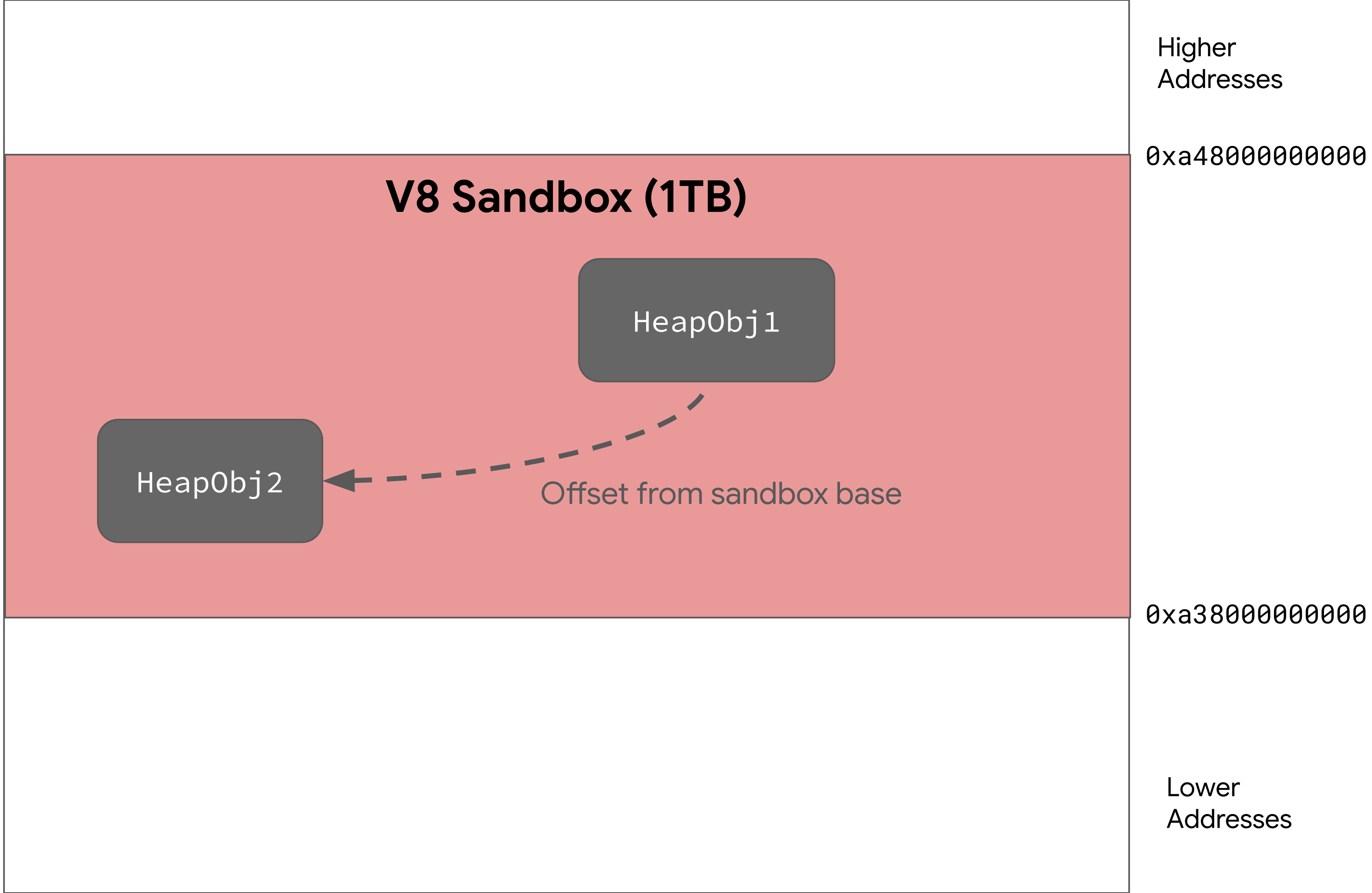
=> Result: an in-process sandbox



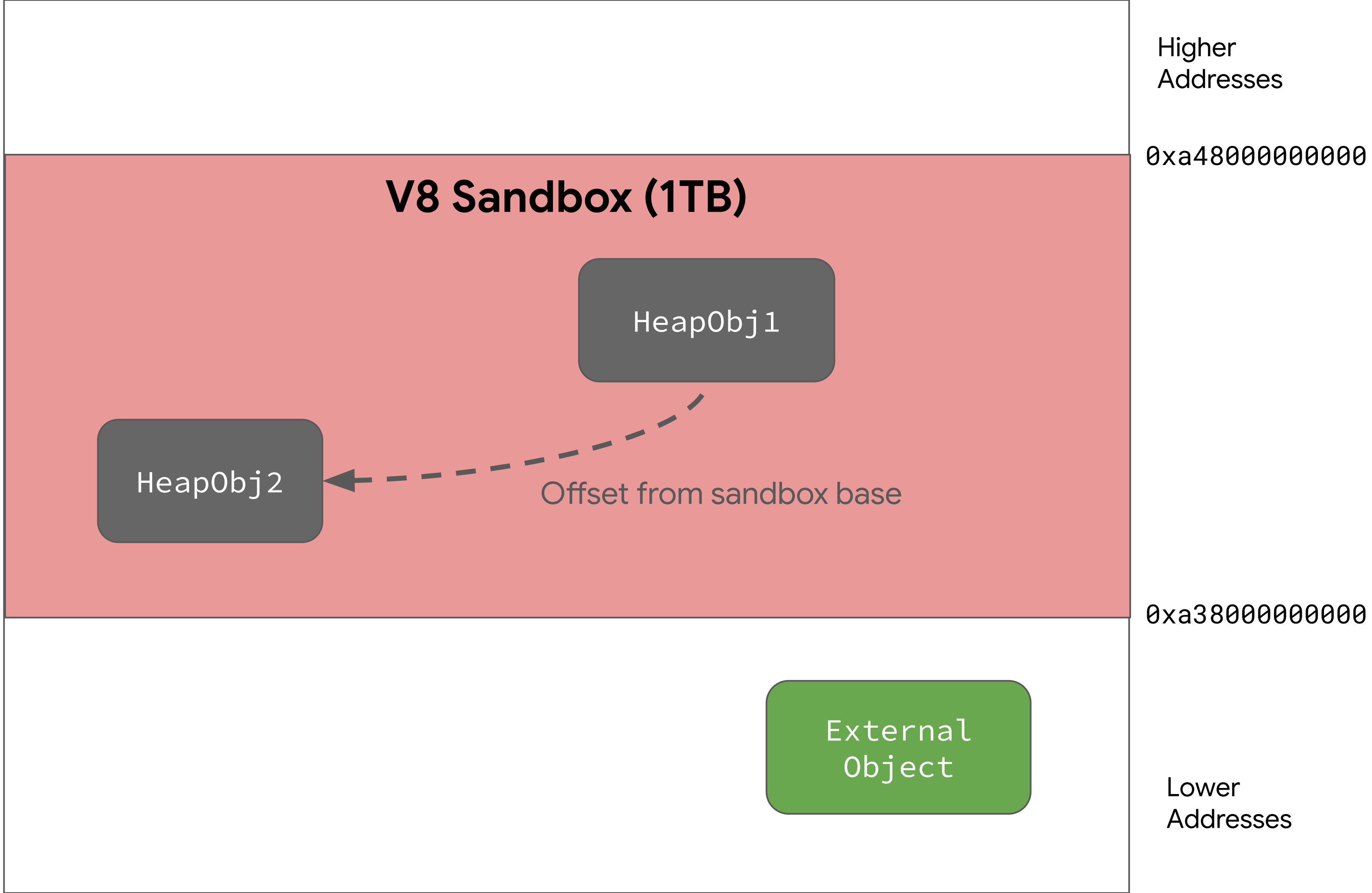

Higher
Addresses
0xa48000000000
V8 Sandbox (1TB)
0xa38000000000
Lower
Addresses

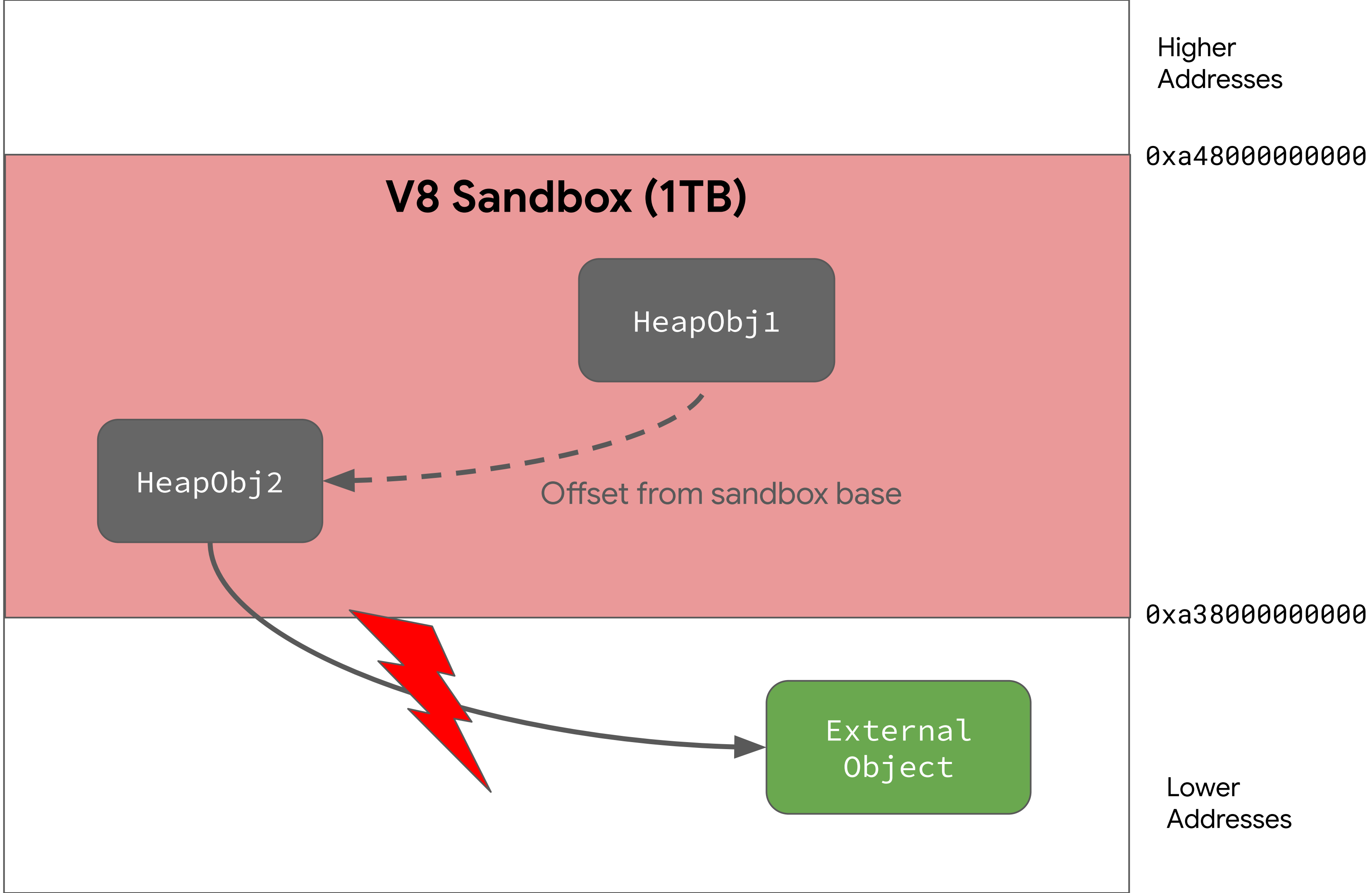

Higher
Addresses
0xa48000000000
V8 Sandbox (1TB)
HeapObj1
HeapObj2
0xa38000000000
Lower
Addresses


Higher Addresses
0xa48000000000
V8 Sandbox (1TB)
HeapObj1
HeapObj2
Raw pointer
0xa38000000000
Lower Addresses

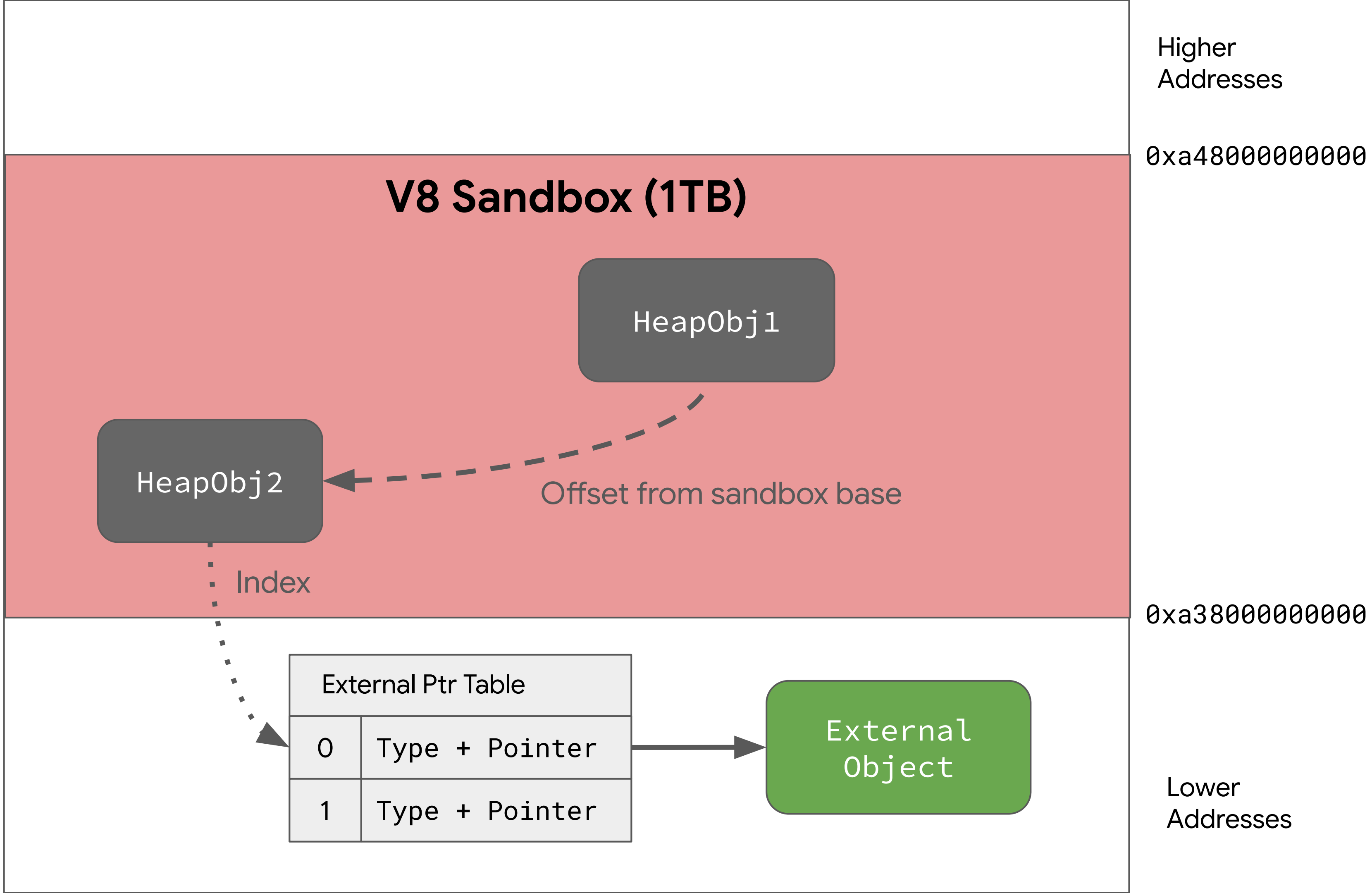


Higher
Addresses
0xa48000000000
V8 Sandbox (1TB)
HeapObj1
HeapObj2
Offset from sandbox base
0xa38000000000
Lower
Addresses

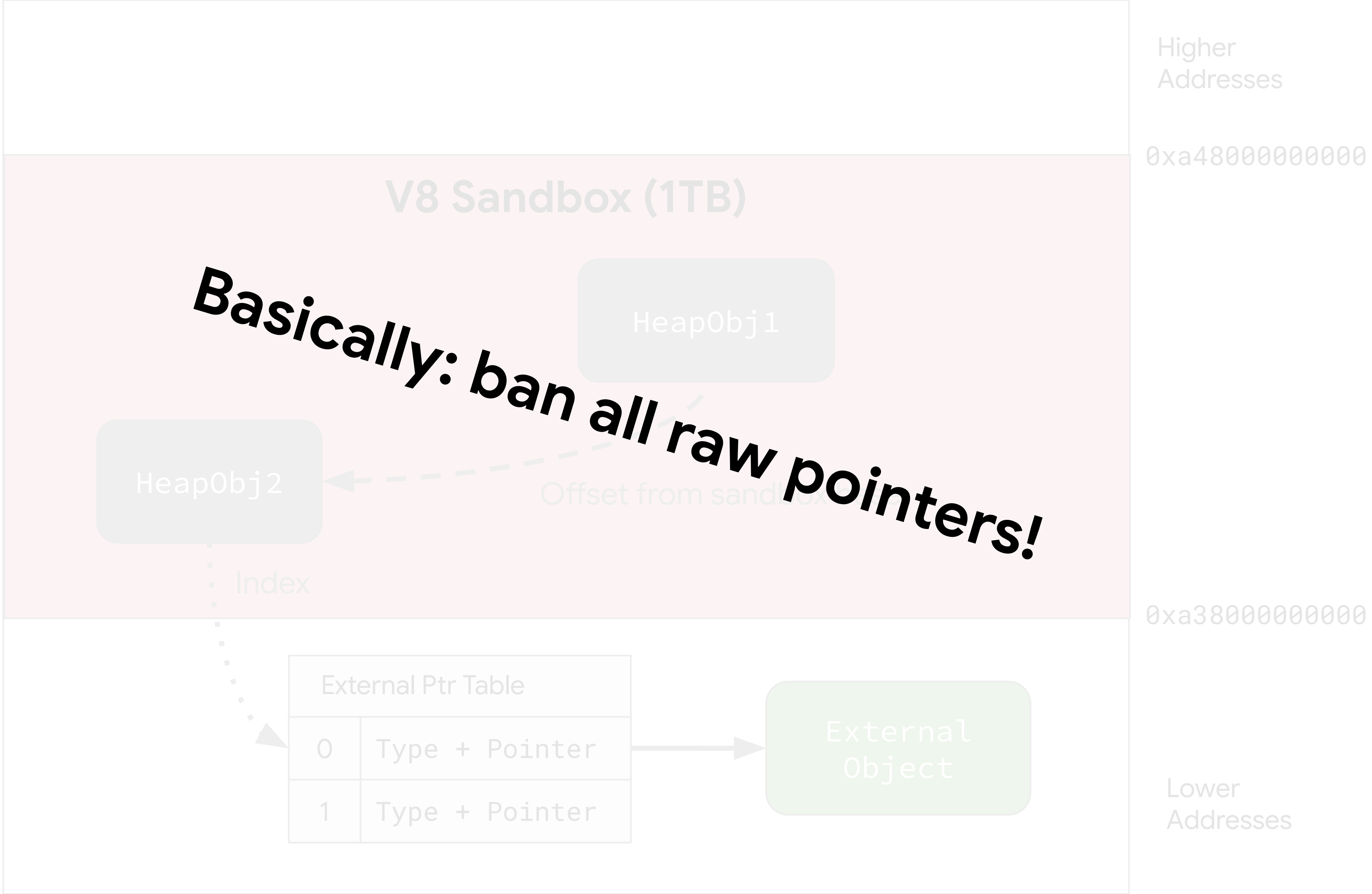

Higher
Addresses
0xa48000000000
V8 Sandbox (1TB)
HeapObj1
HeapObj2
Offset from sandbox base
0xa38000000000
External
Object
Lower
Addresses

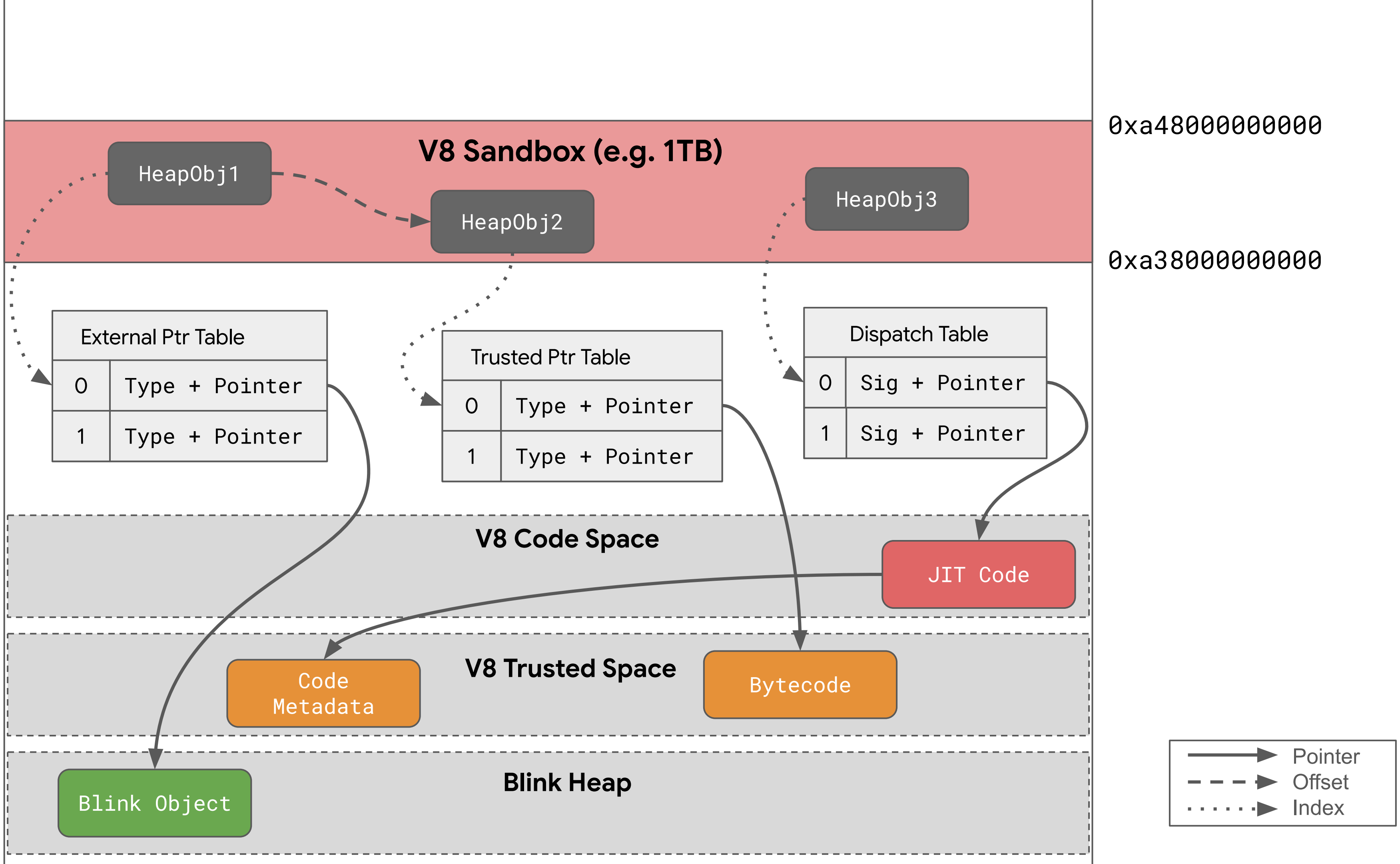


Higher Addresses
0xa48000000000
V8 Sandbox (1TB)
HeapObj1
HeapObj2
Offset from sandbox base
0xa38000000000
External Object
Lower Addresses


Higher Addresses
0xa48000000000
V8 Sandbox (1TB)
HeapObj1
HeapObj2
Offset from sandbox base
Index
0xa38000000000
External Ptr Table
0 Type + Pointer
1 Type + Pointer
External Object
Lower Addresses

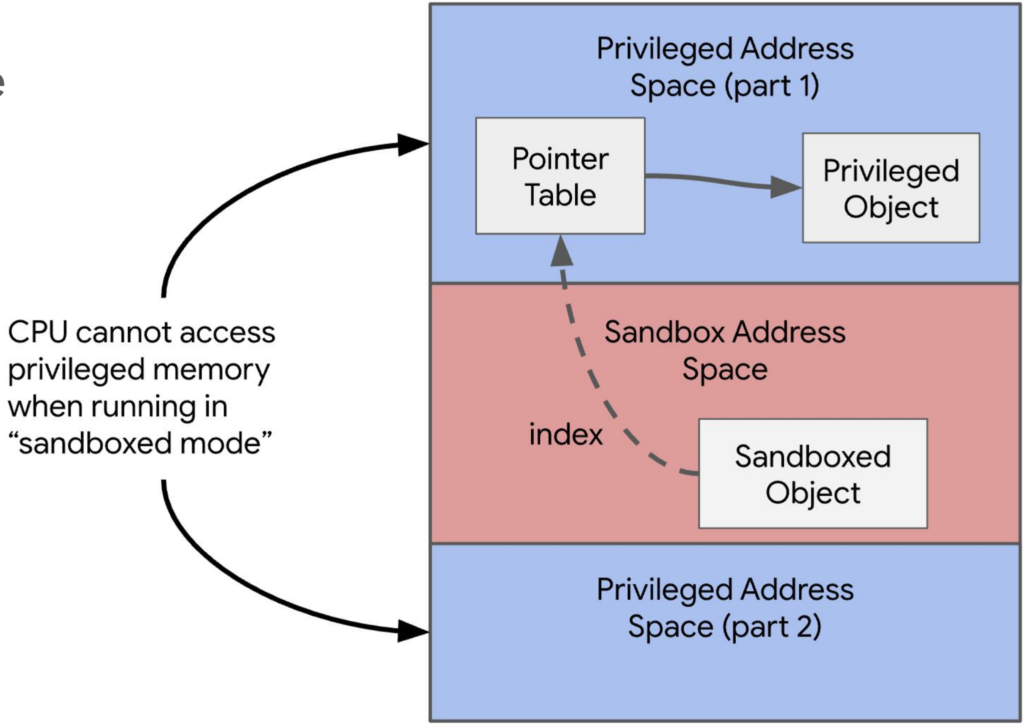
Basically: ban all raw pointers!




0xa48000000000
0xa38000000000
V8 Sandbox (e.g. 1TB)
HeapObj1
HeapObj2
HeapObj3
External Ptr Table
0
Type + Pointer
1
Type + Pointer
Trusted Ptr Table
0
Type + Pointer
1
Type + Pointer
Dispatch Table
0
Sig + Pointer
1
Sig + Pointer
V8 Code Space
JIT Code
V8 Trusted Space
Code Metadata
Bytecode
Blink Heap
Blink Object
Pointer
Offset
Index

Sandbox with Hardware Support?

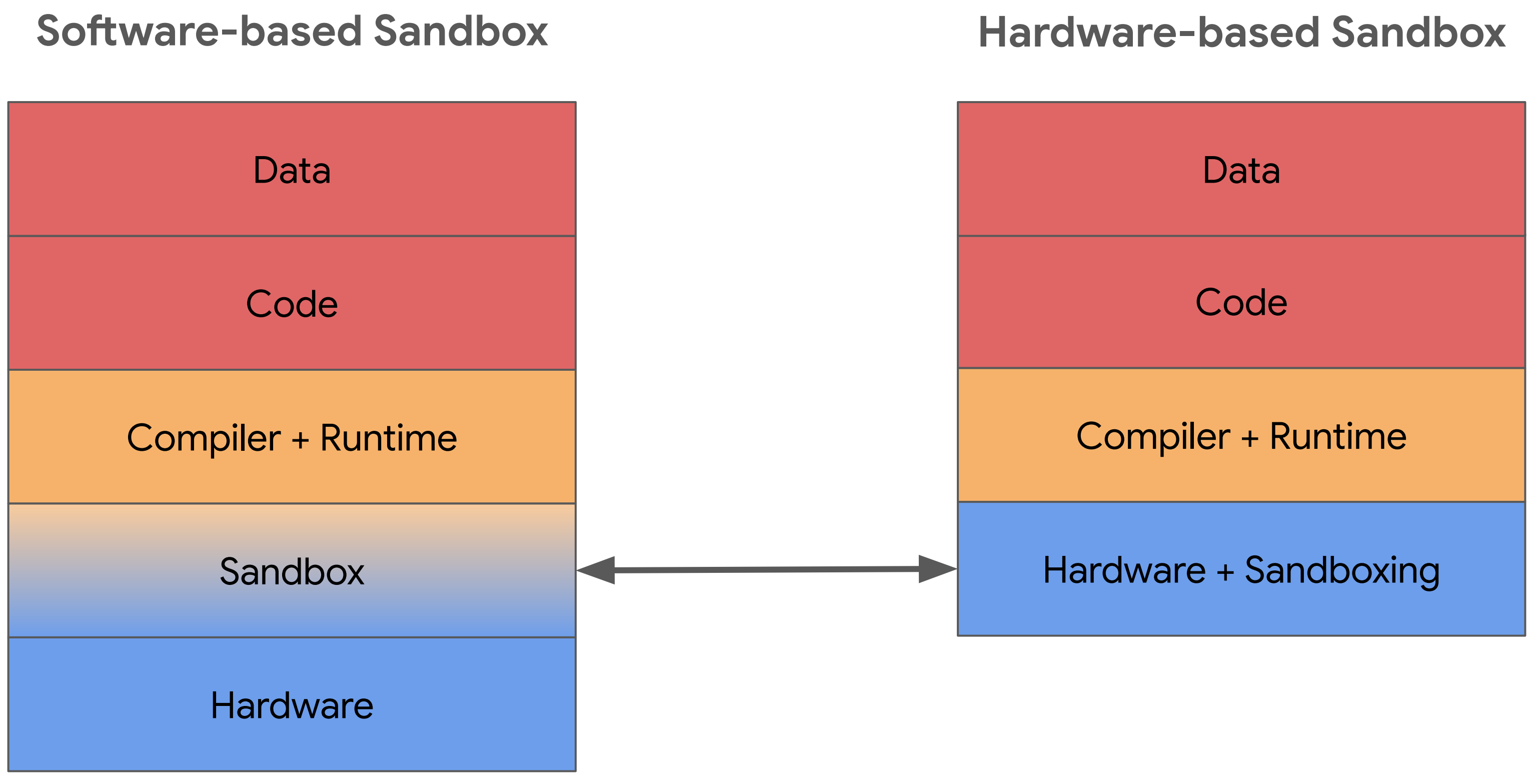
- In the future, should be possible to “drop privileges” when executing JS or Wasm code
- Would be very similar to userspace/kernel split
- Built on top of current software-only sandbox
- Ideally: want to be able to run *untrusted machine code*



V8 Sandbox - “Progress Bar”

- Probably at ~v0.95 of the sandbox (software only)
- Most design-level issues taken care of, but implementation issues remain
- Have a [prototype for hardware sandboxing]() based on Intel PKEYs
    - But PKEYs are fairly limited, [ARM’s upcoming POE2]() is much more powerful! \o/

JavaScript Engine with Sandbox



WebKit's JITCage

- Similar idea: sandbox untrusted code
- Requires special hardware features
- Different goal: prevents JIT-generated code from executing certain instructions (e.g. syscalls) or performing unsafe control-flow transfers
- Similar hardware capabilities available in ARM's upcoming POE2 🎉

JITCage

- **The following instructions can't be executed in the JITCage**
  - RET
  - BR/BLR/BL
  - SVC
  - MRS/MSR

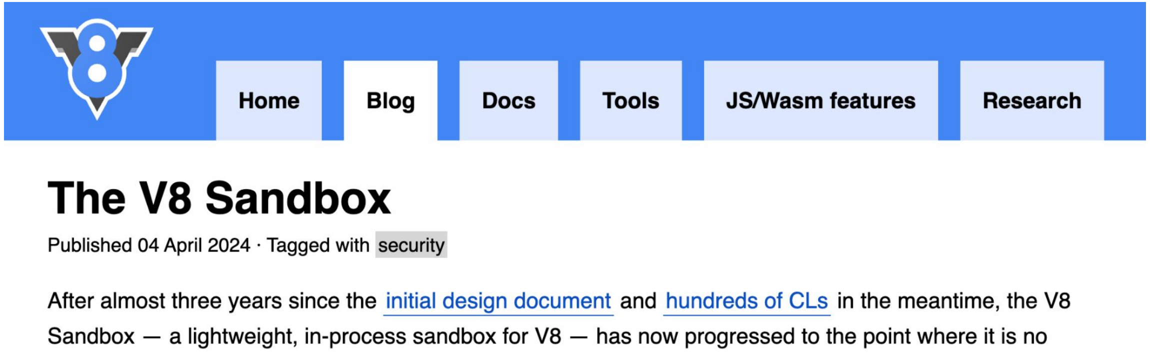
The Future of Sandboxing?

	JITCage	Software Sandbox	Hardware Sandbox
Applicable To	JIT Code (?)	JIT + Builtin Code	JIT + Builtin Code
Restricts Accessible Memory	No	Yes	Yes
Restricts Instruction Set	Yes	Mostly*	Yes
Restricts Control-Flow	Yes	Mostly*	Yes

* Requires additional code validation (think: NaCL) for actual security guarantees

Sandboxing - Summary

- Sandbox is a useful mitigation in itself
    - Attackers already now need multiple bugs (or one really good one) to exploit V8
- But it’s also an ***architecture that enables powerful mitigations***:
    - Memory-safe languages, hardened C++, and/or MTE for privileged code
    - Hardware sandboxing or code validation in software for sandboxed code
- **⇒ Plausible path towards a secure, high-performance JS engine**



Home | Blog | Docs | Tools | JS/Wasm features | Research

The V8 Sandbox

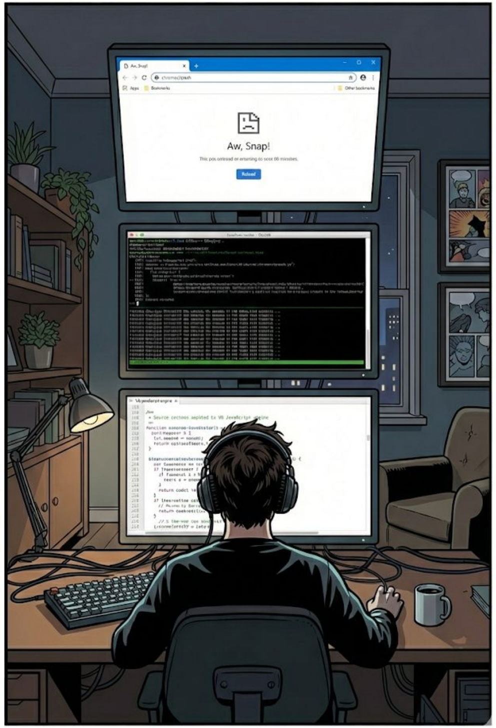
Published 04 April 2024 · Tagged with security

After almost three years since the initial design document and hundreds of CLs in the meantime, the V8 Sandbox — a lightweight, in-process sandbox for V8 — has now progressed to the point where it is no

Modern JS Vulnerability Research

Finding JS Engine Bugs

- The classics approaches are still going strong
- Manual code auditing
  - Time consuming, significant ramp-up work required
  - But allows searching for high-quality vulnerabilities
  - See e.g. some of the awesome bug reports by [Seunghyun Lee (@0x10n)]
- Fuzzing
  - Can find really cool bugs, also less cool ones
  - Targeted fuzzing for e.g. variants or against specific components is still promising



Finding JS Engine Bugs

- But there are also new approaches:
- AI-powered vulnerability research
  - For example [Google's Big Sleep system]
  - Combine latest AI reasoning models with purpose-built tools (code browser and debugger tool) and environment (JS shell)
  - Able to find bugs that fuzzers cannot (or at least struggle a lot with)
  - First V8 bugs found ~mid 2025 (Gemini 2.5 generation)

Bugs found by Big Sleep in Chrome/V8

Chromium

hotlistid:8067258

Sign in

Component Settings

251 - 292 of 292

Hotlist: BigSleep

POS	P	TYPE	TITLE	STATUS	ID
338	P1	Vulnerability	V8: Hole leak due to Maglev's Array Ctor optimization	Fixed	457866804
339	P1	Vulnerability	V8: Out-of-bounds access in JSON.parse	Verified	452296415
340	P1	Vulnerability	V8: Type Confusion in LoadSuperIC	Fixed	450328966
341	P1	Vulnerability	V8: Hole leak in Ignition interpreter due to invalid hole-check removal	Verified	449760249
342	P1	Vulnerability	V8: Integer overflow in TryEmitLoadForLoadWord64AndShiftRight leading to ...	Verified	444049512
343	P1	Vulnerability	V8: Integer truncation during Maglev compilation leading to memory corrupti...	Fixed	444048019
344	P1	Vulnerability	V8: Bytecode corruption due to integer truncation when writing exception han...	Fixed	443765373
345	P1	Bug	V8: Incorrect detection of "fast" keys in JSON parser	Verified	440279755
346	P4	Vulnerability	[BIGSLEEP-440027817] PDFium: out-of-bounds read in PatchDrawer::Draw	Verified	440028542
347	P1	Vulnerability	V8: Type confusion in SourceTextModule::GetCycleRoot	Verified	439986081
348	P3	Vulnerability	V8: ArrayBuffer Use-after-Free in WebAssembly.Validate	Verified	439772737
349	P1	Vulnerability	V8: off-by-one error in RegExp matching leads to OOB read	Fixed	439758498
350	P1	Vulnerability	V8: race condition during TypedArray.sort leads to memory corruption	Verified	439522866

https://issues.chromium.org/issues?q=hotlistid:8067258

Finding JS Engine Bugs

- Also: “Vibe-VR” has really taken off earlier this year
  - Basically: take claude code (or similar), point it at codebase, say “find me a bug”
- Seems to have become viable around the Claude 4.5, Gemini 3 releases
  - ~ late 2025, early 2026
  - IMO less of a raw capability increase, more of an accuracy improvement
  - => reasonably low false-positive rate
  - => huge increase in bug reports

https://bughunters.google.com/blog/evolving-the-android-chrome-vrps-for-the-ai-era

Evolving the Android & Chrome VRPs for the AI Era

The primary goal of the Android & Google Devices and Chrome Vulnerability Reward Programs (VRPs) has been to partner with the external security researcher community to analyze areas lacking coverage, helping us discover vulnerabilities that historically have been hard to find. In today’s shifting security research landscape, we want to continue to reward researchers for their expertise and

Issue 436181695 (Google Big Sleep)

- Bug in bytecode compiler
- Mismatch in number of yield points between parser and compiler
- Leads to jump into the middle of bytecode
  - *Highly* exploitable and also bypasses the (current) sandbox
- Relatively easy to discover but fuzzers had no chance
  - They weren't yet aware of new syntax...

```
async function* bug() {
  for (await using x = { [Symbol.asyncDispose]() {} }; 1; ) {}
}
async function run() {
  for await (const x of bug()) {}
}
run();
```

Issue 443765373 (Google Big Sleep)

- Exception handler is encoded as offset in bytecode
- Offset is stored as 28 bit integer (256 MB)
- Max. bytecode size: 512 MB
- => Huge bytecode array will cause truncation of offset
- => Again arbitrary bytecode execution

```
function simple() {
  try {
    throw 42;
  } catch (e) {
    return e;
  }
}
```

```
[generated bytecode for function: simple]
         0x2ad5001000e4 @    0 : 1b ff f9  Mov <context>, r0
         0x2ad5001000e7 @    3 : 0d 2a     LdaSmi [42]
         0x2ad5001000e9 @    5 : b5        Throw
         0x2ad5001000ea @    6 : d1        Star1
         0x2ad5001000eb @    7 : 8d f8 00  CreateCatchContext r1, [0]
         0x2ad5001000ee @   10 : d2        ...
         ...
         0x2ad5001000f7 @   19 : b7        Return
Handler Table (size = 16)
   from    to       hdlr (prediction,   data)
  (   3,    6)  ->     6 (prediction=1, data=0)
```

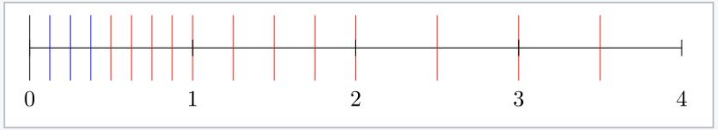
Issue 382005099 (V8 Team)

- Bad interaction between WebAudio and JavaScript engine
- WebAudio supports custom audio processing nodes defined in JS
- For performance reasons, WebAudio changes CPU's handling of floats
- ⇒ JavaScript code runs in unexpected CPU mode
- ⇒ Turns out this is exploitable
- Affected multiple browsers (e.g. also CVE-2025-24213 in Safari)
- **Quite possibly my favorite bug of the last few years :)**

Background: Denormal Floats

- Floating point number: 1 sign bit, X mantissa bits, Y exponent bits
  - Final value: `(-1)**sign * (1+mantissa) * 2**(exponent-bias)`
- Normal float numbers: no leading zero bits in mantissa
- Denormal float number: leading zero bits in mantissa
- For better performance: CPU can disable denormals ⇒ they become zero
- BUT: JavaScript spec assumes denormals are supported and != zero

In computer science, **subnormal numbers** are the subset of **denormalized numbers** (sometimes called **denormals**) that fill the underflow gap around zero in floating-point arithmetic. Any non-zero number with magnitude smaller than the smallest positive normal number is *subnormal*, while *denormal* can also refer to numbers outside that range.[1]



An unaugmented floating-point system would contain only normalized numbers (indicated in red). Allowing denormalized numbers (blue) extends the system's range.

https://en.wikipedia.org/wiki/Subnormal_number

Issue 382005099 (trigger)

```
const denormal = 5E-324;

console.log(`Denormal float value outside processor: ${denormal}`);
// Prints "5e-324"

class DenormalDemoProcessor extends AudioWorkletProcessor {
  process(inputs, outputs, parameters) {
    this.port.postMessage(`Denormal float value inside processor: ${denormal}`);
    // Prints "0" (!)
  }
}

registerProcessor('denormal-demo-processor', DenormalDemoProcessor);
```

Issue 382005099 (PoC Exploit)

```
// See https://crbug.com/382005099#comment19

function poc(x) {
  let obj = {denormal: 5E-324};
  new Float64Array();

  let positive = (x & 1) + 1;
  let denormal = Math.min(obj.denormal, positive);
  let b = Object.is(denormal, 0);

  let n = b | 0;
  n *= 0xffffffff;
  let o = n + 1;

  let o_ = (Math.random() <= 1) ? o : undefined;
  let i = Math.sign(o_) * 64;
  let first = [1];
  first[i] = 2;

  let second = [1,2,3];
  return {first, second};
}
```

- Exploit abuses assumptions made by optimizing compiler
  - Essentially that `5E-324 != 0`
- Leads to incorrect range analysis
- Requires a few more tricks to make it work though
- Might also be exploitable in other ways, e.g. via bytecode compiler

Thank you!

Questions?