V8 Sandbox

Aka. “Ubercage”

Author: saelo@
First Published: July 2021
Last Updated: December 2023
Status: Living Doc
Visibility: **PUBLIC**

This document is part of the V8 Sandbox Project and covers the high-level design of the sandbox.

Summary

Objective: build a low-overhead, in-process sandbox for V8.

Motivation: V8 bugs typically allow for the construction of unusually powerful and reliable exploits. Furthermore, these bugs are unlikely to be mitigated by memory safe languages or upcoming hardware-assisted security features such as MTE or CFI. As a result, V8 is especially attractive for real-world attackers.

Design: The proposal assumes that an attacker can arbitrarily corrupt memory on the V8 heap, where JavaScript objects are located. This primitive can be constructed from typical V8 vulnerabilities. To protect other memory within the same process from corruption, and by extension to prevent the execution of arbitrary code, the V8 heap is moved into a pre-reserved region of virtual address space: the sandbox. Then, all memory accesses performed by V8 must either be restricted to the sandbox address space (e.g. by using offsets instead of raw pointers to reference objects) or be validated in some way (e.g. by using a pointer table indirection). In particular, full 64-bit pointers must be banned entirely from the V8 heap.

Objective

Build an in-process sandbox for V8 to prevent an attacker who successfully exploited a V8 vulnerability, and thus is able to corrupt objects inside the V8 heap, from corrupting other memory in the process and thus from executing arbitrary code. In essence, this will turn arbitrary writes originating from V8 vulnerabilities into bounded writes. Preventing reads (direct or speculative) outside of the V8 heap is not a goal, however. The performance overhead should be minimal, with a rough target of around 1% overall on real-world workloads. This sandbox should eventually become a supported security boundary.

Motivation

Many V8 vulnerabilities exploited by real-world attackers are effectively *2nd order vulnerabilities*: the root-cause is often a logic issue in one of the JIT compilers, which can then be exploited to generate vulnerable machine code (e.g. code that is missing a runtime safety check). The generated code can then in turn be exploited to cause memory corruption at runtime. This appears to be a somewhat natural problem of JIT compilers for dynamic languages, as one of their major purposes is to remove (redundant) runtime checks that would otherwise be performed by the interpreter.

As an example, consider the case of a typical JIT compiler incorrect side effect modeling vulnerability: here, the compiler will model the side effects of an operation incorrectly and can then be tricked into emitting machine code that is lacking a runtime type check after such an operation (because the compiler believes that the object could not have changed its type during the operation). As such, the emitted machine code is now vulnerable to a type confusion, and the attacker can exploit that to cause (fairly arbitrary) memory corruption at runtime. These types of issues are uniquely attractive for attackers for a number of reasons:

- The attacker has a great amount of control over the memory corruption primitive and can often turn these bugs into highly reliable and fast exploits
- Memory safe languages will not protect from these issues as they are fundamentally logic bugs
- Due to CPU side-channels and the potency of V8 vulnerabilities, upcoming hardware security features such as memory tagging will likely be bypassable most of the time

Due to the nature of these vulnerabilities, and their uniqueness to JavaScript engines, it seems desirable to build a custom sandboxing mechanism for V8.

Attacker Model

This proposal assumes that an attacker is capable of repeatedly performing arbitrary reads and writes inside the V8 heap as well as potentially performing reads outside of the V8 heap, for example due to speculative side-channel attacks. This reflects common initial exploitation primitives gained from many V8 vulnerabilities, such as vulnerabilities in the JIT compilers, the runtime, or the garbage collector.

The ability to corrupt memory outside of the V8 heap is then considered to be an escape from this sandbox. This definition also covers arbitrary code execution.

Design

Since early 2020, V8 has implemented pointer compression in its heaps. With pointer compression, every reference from an object in the V8 heap to another object in the V8 heap ("on-heap") becomes a 32 bit offset from the base of the heap, leaving only a few objects with raw pointers to objects outside the v8 heap ("off-heap"). Compressed pointers are then only valid inside a 4GB virtual memory region, referred to as the pointer compression cage. Pointer compression can be visualized with an instance of an ArrayBuffer in memory. The following shows the in-memory layout of a hypothetical ArrayBuffer object *without* pointer compression:

```
(lldb) x/6gx 0x2507080c5f70
0x2507080c5f70: 0x0000250708281181 0x00002507080406e1
0x2507080c5f80: 0x00002507080406e1 0x0000000000001000
0x2507080c5f90: 0x0000000107845c00 0x0000000107632708
```

The ArrayBuffer object contains the following fields:

- Map (pink), 64 bit on-heap pointer
- JS Properties array (blue), 64 bit on-heap pointer
- JS Elements array (red), 64 bit on-heap pointer
- Size in bytes (magenta), 64 bit unsigned integer
- Backing storage (purple), 64 bit off-heap pointer
- ArrayBufferExtension (orange), 64 bit off-heap pointer

With pointer compression enabled, the same ArrayBuffer object would be stored as shown next:

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

Here, all on-heap pointers were converted to 32-bit compressed pointers. In this example, the heap base would be 0x250700000000 and so for example the compressed Map pointer (0x08281181) would be interpreted as the absolute pointer 0x250708281181.

The majority of V8 vulnerabilities can be exploited to corrupt memory (only) in the V8 heap (out-of-bounds accesses, type confusions, et al). With pointer compression enabled, an attacker with the ability to corrupt data in the V8 heap gains no additional capabilities by corrupting a compressed pointer. Instead, to reach outside the V8 heap, the attacker targets one of the remaining raw pointers in the heap (typically an ArrayBuffer or TypedArray backing store pointer). The fundamental idea behind this project is to protect ("sandboxify") all remaining raw

pointers in a way that prevents their abuse by an attacker. On a high level, this is achieved in the following way:

- A large (for example 1TB) region of virtual address space - the sandbox - is reserved during initialization of V8. This region contains the pointer compression cage, and so all V8 heaps, as well as ArrayBuffer backing stores and similar objects.
- All objects inside the sandbox, but outside of V8 heaps, are addressed using fixed-size offsets (e.g. 40-bit offsets in the case of a 1TB sandbox) instead of raw pointers.
- All remaining off-heap objects must be referenced through a pointer table, which contains the pointer to the object together with type information to prevent type confusion attacks. Entries in this table are then referenced from objects in the v8 heap through indices.

With this, the ArrayBuffer object from above would become:

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

Here, the raw, off-heap backing storage pointer (purple) has been replaced with a 40-bit offset from the base of the sandbox (the offset is 0x45c00, shifted to the left by 24 to guarantee that the top bits are zero). On the other hand, the raw pointer to the ArrayBufferExtension object (orange) has been replaced with a 32-bit index into a pointer table. In this example, the size (magenta) remains unchanged, but would be verified on access to be smaller than the maximum allowed size of an ArrayBuffer. Alternatively, it could also be stored shifted to the left as well.

Attackers are now assumed to be able to corrupt memory inside the sandbox arbitrarily and from multiple threads, and will now require an additional vulnerability to corrupt memory outside of it, and thus to execute arbitrary code. However, the attack surface of the sandbox will likely be significantly less complex than V8 itself due to the relatively low complexity of the embedder <-> V8 interface, and bugs in the sandbox appear to mostly be “classic” memory corruption bugs as opposed to bugs in V8. For these reasons, the sandbox is assumed to be an easier-to-defend security boundary than the V8 VM.

Finally, it is worth noting that, although not an explicit goal, this design also prevents an attacker from directly (but not speculatively) reading, not just writing, data outside of the sandbox.

The remainder of this section discusses the central sandboxing mechanisms in some more detail, then concludes with a brief summary of the design.

Sandbox Address Space

The sandbox currently assumes a shared pointer compression cage which is shared between all V8 Isolates and thus Heaps in the process. This 4GB region is then placed at the start of a much larger (e.g. 1TB) virtual address space reservation - the sandbox - which is surrounded by large guard regions on both sides to prevent indexed accesses to reach outside of the sandbox. The remaining space in the sandbox is used to allocate other objects directly referenced by V8, such as ArrayBuffer backing stores and the 10GB Wasm memory cages.

The address space reservation backing the sandbox is discussed in more detail in this document.

Sandboxed Pointers

All objects located inside the sandbox can be referenced from objects in the V8 heaps through a 40-bit (in the case of a 1TB sandbox) offset from the base of the sandbox. These are called “SandboxedPointers”. Using an offset instead of a raw pointer prevents an attacker from addressing memory outside of the sandbox.

SandboxedPointers are especially performant as the base address of the sandbox is usually already available in a register. The offsets can then be stored shifted to the left, in which case loading a sandboxed pointer only requires shifting the loaded value to the right and adding it to the base register. This is possible with two additional instructions on x64 (shift + add) and a single additional instruction on arm64 (the add instruction can also perform the shifting).

An attacker able to corrupt data inside a V8 heap can then corrupt the offsets (and sizes) of ArrayBuffer objects, and so it must be assumed that an attacker can corrupt any data inside the sandbox. However, a SandboxedPointer guarantees that it will always point into the sandbox.

Sandboxed pointers are discussed in more detail in this document.

Pointer Tables

All objects located outside the sandbox (“external entities”) are referenced through pointer tables, which are themselves also located outside of the sandbox. This is conceptually similar to the file descriptor table used by an operating system’s kernel or a Wasm table.

In general, the sandbox differentiates between three different types of external entities:

- Executable code, which is managed by V8’s garbage collector but allocated in a dedicated code space, located outside of the sandbox. These are referenced via a CodePointerTable (CPT), discussed in more detail in this document.

- V8 objects managed by V8's garbage collector but located outside of the sandbox for security reasons. See the Trusted Objects section below and this document for more details. These are referenced via a TrustedPointerTable (TPT).
- Non-V8 objects that are not directly managed by V8's garbage collector. These include all embedder-managed objects such as DOM nodes as well as the ArrayBufferExtension object from the example above and many other types of objects. These are referenced via an ExternalPointerTable (EPT), which is discussed in more detail in this document.

This section now briefly discusses how those objects are protected, in particular how memory safety is achieved. For a more detailed discussion of these mechanisms, refer to the design documents linked above.

Temporal Memory Safety

Entries in a pointer table are managed by the garbage collector. If an entry is no longer referenced from any object in the V8 heap, the entry is cleared and the pointed-to object is released if it is managed by V8 (unless there are other references to it from elsewhere). Any subsequent access to the entry will then either see an invalid pointer if the entry is still free, or see a valid pointer to a live object if it was reused. In the latter scenario, the access is safe by design if the new object is of the same type as the previous one, otherwise, the access would fail due to the type safety mechanism described below.

Spatial Memory Safety

Some objects such as ExternalStrings reference a buffer of data of a given length. This sandbox must ensure that any access to those external buffers stays in bounds of the allocated memory. This is generally possible in two ways: (1) by storing length information in the table or the external object itself and bounds-checking against that or (2) by moving those buffers into the sandbox.

Type Safety

To ensure type safety of external objects, the table entries consist of pairs of pointers and type tags (with the type tags potentially stored in unused pointer bits for efficiency). Before an external object is accessed, the type tag of the entry must be checked against the caller-supplied expected type, either with an explicit check or by ensuring that wrong types will result in an inaccessible address.

Thread Safety

The sandbox has to prevent concurrent access to external objects that aren't thread safe. Ideally, this will be achieved by having a dedicated pointer table per Isolate and thus per mutator thread and ensuring that a non-thread-safe external object is only referenced from at most one table.

Trusted Objects

There are a number of internal V8 objects that do not contain pointers, but which could still allow an attacker to break out of the sandbox. One example are code-like objects, such as interpreter bytecode, JIT-compiled machine code, and any related metadata. These are not generally robust against corruption and will thus allow an attacker to escape from the sandbox. Other examples could include heap allocator metadata or V8 objects that contain indices into off-heap data structures.

In general, for the sandbox to become robust, these objects either need to move out of the V8 heap into a “trusted” heap space, become robust against corruption, be treated as untrusted (and have some way of checking their integrity on access), or be marked as read-only.

A generic mechanism for moving such objects out of the sandbox and referencing them via a pointer table is discussed in this document.

Summary

Conceptually, the heap sandbox design is summarized in the diagram below (guard regions around the sandbox omitted).