V8 Heap Sandbox

Aka. “Ubercage”

Author: saelo@
First Published: July 2021
Last Updated: July 2021
Status: Draft
Visibility: **PUBLIC**

This document is part of the V8 Heap 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 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 proposed sandbox assumes that an attacker can arbitrarily corrupt memory inside the V8 heap, a primitive gained from a typical V8 vulnerability. To protect other memory in the same process from corruption, and by extension prevent the execution of arbitrary code, all raw pointers in the V8 heap are converted into either offsets into a virtual memory “cage” or indices into an external pointer table.

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 memory outside of the V8 heap and thus from executing arbitrary code. In essence, this will turn arbitrary writes originating from V8 vulnerabilities into bounded writes. The performance overhead should be minimal, with a rough target of around 1% overall on real-world workloads. Ideally, this sandbox would 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 modelling 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 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, absolute pointers to objects outside the v8 heap ("off-heap"). Compressed pointers are then only valid inside a 4GB virtual memory region. 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 0x2507080c5f7c
0x2507080c5f78: 0x0000250708281181 0x00002507080406e1
0x2507080c5f80: 0x00002507080406e1 0x0000000000001000
0x2507080c5f88: 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 uncompressed pointers in the heap (typically an ArrayBuffer or TypedArray backing

store pointer). The fundamental idea behind this project is to protect the remaining off-heap pointers in a way that prevents their abuse by an attacker. For that, there are two central mechanisms:

1. A virtual memory cage which contains both the 4GB V8 heap region and, following that, a large (likely hundreds of GB up to a few TB) region containing pure data buffers such as ArrayBuffer backing stores and WASM memory cages. These data buffers can then be referenced from objects in the V8 heap through an offset rather than a raw pointer.
2. An external pointer table, in which pointers to all remaining types of off-heap objects are stored together with type information to prevent type confusion attacks. Entries in this table are then referenced from objects in the v8 heap through indices (conceptually similar to the file descriptor table provided by an OS or a WASM table).

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 virtual memory cage (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 the external 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.

With this sandbox, attackers are assumed to be able to corrupt memory inside the virtual memory cage 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 reading (not just writing) data outside of the virtual memory cage.

The remainder of this section first describes the virtual memory cage and the external pointer table in some more detail, then discusses other data structures that need to be protected for the sandbox to become robust, and finally concludes with a brief summary of the design.

Virtual Memory Cage

The virtual memory cage currently assumes a shared pointer compression cage: all V8 heaps share the same 4GB virtual memory region. This 4GB region is then placed at the start of a much larger (e.g. 1TB) virtual address space reservation - the virtual memory cage - which is surrounded by large guard regions on both sides.

ArrayBuffer backing stores and similar pure-data buffers are placed inside the cage and can then be referenced from objects in the V8 heaps through a 40-bit (in the case of a 1TB cage) offset from the start of the cage while also limiting their maximum size. This is especially performant as the base address of the cage is usually stored in a register. The offsets can then be stored shifted to the left, in which case uncaging such a 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 virtual memory cage.

External Pointer Table

All references to objects located outside the virtual memory cage (“external objects”) are kept in external pointer tables, which are themselves located outside of the cage. This section briefly discusses how those objects are protected, in particular how memory safety is achieved.

Temporal Memory Safety

Entries in the external 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 (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 virtual memory cage.

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 external 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 Data Structures

There are a number of data structures currently located inside the V8 heap that do not contain pointers, but which could still allow an attacker to break out of the sandbox. One example of such data structures are code-related 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 data structures either need to move out of the V8 heap and thus become external objects, become robust against corruption, be treated as untrusted (and have some way of checking their integrity on access), or be marked as read-only.

Summary

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