V8 Sandbox - Hardware Support

Author: saelo@
First Published: February 2024
Last Updated: February 2024
Status: Living Doc
Visibility: **PUBLIC**

This document is part of the V8 Sandbox Project and discusses different options for how dedicated hardware support could be used to strengthen or even replace the software-based sandbox. For a general overview of the (software-based) sandbox design see the high-level design document.

Background

The sandbox strives to build a privilege boundary within the process hosting V8 (e.g. a Chromium renderer process): untrusted code - for example the JavaScript code executed by V8, or even parts of V8 itself - should run with lower privileges than the rest of the process. In the software-based sandbox, this is achieved indirectly: untrusted code only operates on data inside the sandbox, and all data stored there must be “sandbox-compatible”, so cannot for example contain raw pointers that would allow accessing data outside of the sandbox.

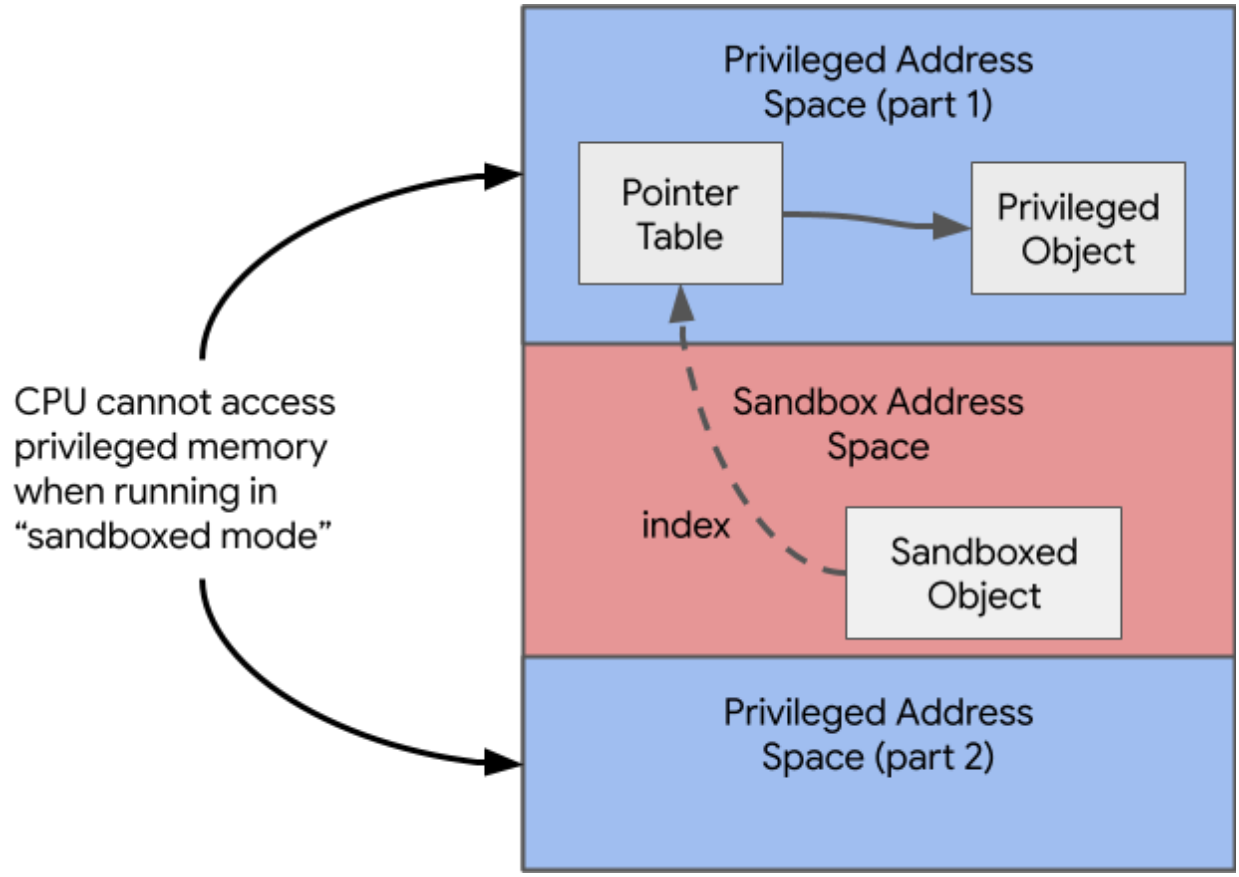
This privilege separation is not unlike the kernel-userland split used by modern operating systems. However, there it is enforced through hardware: the CPU runs at a lower privilege level when executing userspace code, which makes it impossible to access memory belonging to the kernel. This document explores whether similar hardware-based approaches could work for the V8 sandbox as well.

Throughout the rest of this document, the words “trusted”/”privileged”/”unsandboxed” and “untrusted”/”unprivileged”/”sandboxed” will be used mostly interchangeably.

Design

A hardware-supported sandbox would not be fundamentally different from a purely software-based one: the core idea remains to partition the address space into trusted and untrusted regions, and using pointer table indirections for references that cross the trust boundary (at least from unprivileged to privileged) as shown in the picture below. Then, when executing “sandboxed” code, the CPU would be explicitly put into a low-privilege mode in

which it cannot write to privileged memory (the sandbox only attempts to prevent corruption, not disclosure, of trusted memory):



The design for a hardware-supported sandbox then needs to answer two central questions:

1. which *data* is considered privileged or unprivileged
2. which *code* is considered privileged or unprivileged

The software-based sandbox has a clear answer for question 1: all memory inside the sandbox address space is considered untrusted, everything else is trusted. However it does not (need to) precisely answer question 2 as there are no explicit `enter_sandbox`/`leave_sandbox` operations. Instead, most code operating on the data inside the sandbox is in some sense considered to be sandboxed. The below table summarizes the software-based sandbox design in terms of these two questions.

Data	Code
Main V8 Heap (most V8 objects)	JavaScript Code (interpreted or JIT-compiled)

Trusted V8 Heap (e.g. bytecode)	WebAssembly Code
Stack and Registers	V8's Runtime Code (e.g. the builtins)
V8/Chromium binary and library	V8's Garbage Collector
Compiler data (mostly ZoneAllocator)	V8's Compilers
Other Heaps (system heap, Blink heap, ...)	Embedder Code (all of Blink)

Here, red means "sandboxed" and blue means "unsandboxed/trusted" while yellow means that the code is implicitly (and maybe only partially) considered to be sandboxed as it mostly operates on data inside the sandbox. Note that while V8's compilers are considered trusted (as they mostly operate on data *outside* of the sandbox), their output (the generated code) is not, which is what matters in practice.

With hardware support, question 2 (code) will need to be answered precisely. Further, its answer to question 1 (data) may differ from that of the software-based sandbox. Due to the many different data- and code regions, there are many possible options for the design of a hardware-supported sandbox. Broadly, these options can be divided into "hardware-assisted" sandboxing, which keeps all the software mechanisms and restrictions in place, and "hardware-based" approaches which replaces at least some of them. Both of these will be discussed next.

Hardware-Assisted Sandbox

With a hardware-assisted sandbox, all mechanisms on which the software-only sandbox relies would be kept in place. In addition, the CPU would enter a low-privilege mode before executing untrusted code.

As such, a hardware-assisted sandbox would mostly answer question 1 (data) in the same way as the software-based sandbox does. In the simplest case, the answer to question 2 (code) would be to treat all JIT-generated code as untrusted. Alternatively, and maybe preferably, all code generated by V8's compilers, *including ahead-of-time compiled code* (e.g. CSA code) would be considered untrusted. In that case, numerous builtins and the entire JavaScript interpreter would run inside the sandbox. The benefit of this approach is that the mode switching would happen exactly at the transitions between C++ and non-C++ code, which are already well-defined through trampolines.

The following table summarizes the hardware-assisted sandbox in terms of the two central questions:

Data	Code
Main V8 Heap (most V8 objects)	JavaScript Code (interpreted or JIT-compiled)
Stack and Registers	WebAssembly Code
Trusted V8 Heap (e.g. bytecode)	V8's Runtime Code (e.g. the builtins)
V8/Chromium binary and library	V8's Garbage Collector
Compiler data	V8's Compilers
Other Heaps (system heap, Blink heap, ...)	Embedder Code (all of Blink)

As can be seen, the main difference to the software-based sandbox is that the JavaScript and WebAssembly code executed by V8 are now explicitly considered untrusted and are executed in a low-privileged mode. In this mode, only the sandbox (and the stack, see below) are writable, everything else is read-only (or not accessible at all). This design would mitigate certain bugs such as the use of 64-bit sizes inside the sandbox or accidental sign-extension when computing indices, which could also result in a memory access outside of the sandbox. Note that both the JavaScript interpreter and JIT-generated code require access to the stack, but the stack cannot be moved into the sandbox (as it contains full pointers). As such, the stack must be accessible in sandboxed mode, and is therefore marked as partially untrusted (yellow) in the table above.

Another way to visualize this sandboxing mode is to look at a typical stack trace. Below is an abbreviated stack trace captured when allocating a new V8 object (in this case, an `ArrayBuffer`):

```
#0  v8::internal::JSArrayBuffer::Setup
#3  0x000055555af1b183 in v8::internal::Builtin_ArrayBufferConstructor
#4  0x000055555ec1ad7d in Builtins_CEntry_Return1_ArgvOnStack_BuiltinExit ()
#5  0x000055555e89d05d in
Builtins_InterpreterPushArgsThenFastConstructFunction ()
#6  0x000055555f25321f in Builtins_ConstructHandler ()
#7  0x000055555e89bd0f in Builtins_InterpreterEntryTrampoline ()
#8  0x000055555e89bd0f in Builtins_InterpreterEntryTrampoline ()
#9  0x000055555e89bd0f in Builtins_InterpreterEntryTrampoline ()
#10 0x000055555e8928dc in Builtins_JSEntryTrampoline ()
#11 0x000055555e892607 in Builtins_JSEntry ()
#15 0x000055555adcdd7d in v8::Script::Run
#17 0x000055555ad2e42d in v8::Shell::ExecuteString
```

```
#21 0x000055555ad4af2d in v8::Shell::Main
#22 0x000055555ad4b522 in main
```

Here, the CPU enters “sandboxed mode” when it starts executing JavaScript code (via the `JSEntry` builtin), in this case via the interpreter (but it could also be JIT-compiled code). It then leaves sandboxed mode either when returning, or when executing builtin code implemented in C++. In this example, the ArrayBuffer constructor is implemented in C++, and so the `CEntry` trampoline is executed, at which point the CPU would leave the sandboxed mode. Similar transitions would happen for many other common tasks: calling out to the embedder (for example, into Web APIs in Blink), performing common object modification tasks such as installing or modifying properties, or notifying the garbage collectors of changes to the heap graph (e.g. via write barriers). As such, this design is very performance sensitive, and ultimately the choice of whether to implement it will depend on the performance overhead of the transitions and whether the security benefits are deemed sufficiently high.

Pros:

- Simple to implement: all that would be necessary is to mark sandbox and non-sandbox pages and enter the low-privilege mode when transitioning between C++ and V8-generated code.
- Strictly an improvement in terms of security guarantees.
- Can mitigate certain vulnerabilities that would otherwise allow breaking out of the sandbox. For example the use of 64-bit sizes inside the sandbox or some “low-level” compiler bugs such as register allocation issues.

Cons:

- Performance sensitive: mode switching will happen very frequently as there will be many transitions into and out of sandboxed code.
- Even though the stack is considered trusted, it must still be writable by sandboxed code.
- Likely impossible to sandbox components such as the garbage collector or the compilers themselves as they are written in C++. However, it’s somewhat unclear if there would be much benefit in doing so.

Hardware-Based Sandbox

In contrast to a hardware-assisted sandbox, a hardware-based sandbox would remove or replace some of the restrictions and mechanism of the software-based sandbox such that the hardware support becomes a necessity, rather than a defense-in-depth measure. In particular, it would again allow storing full 64-bit pointers inside the sandbox in some cases.

An initial step in this direction would be to add a dedicated “sandbox stack” that is allocated inside the sandbox and used when executing sandboxed code (i.e. there would also be a stack switch at the privilege boundary). As that stack would contain raw 64-bit pointers, this mode would no longer be safe without the hardware sandbox mode. With this “simple” hardware-based sandbox, the two questions from above would be answered as follows:

Data
Main V8 Heap (most V8 objects)
Untrusted stack (inside the sandbox)
Trusted V8 Heap (e.g. bytecode)
Trusted Stack
V8/Chromium binary and library
Compiler data
Other Heaps (system heap, Blink heap, ...)

Code
JavaScript Code (interpreted or JIT-compiled)
WebAssembly Code
V8’s Runtime Code (e.g. the builtins)
V8’s Garbage Collector
V8’s Compilers
Embedder Code (all of Blink)

Once established, more and more code and data could gradually be moved into the hardware-based sandbox, until potentially all of V8 is effectively sandboxed. However, it remains an open question whether this is actually practically feasible, and how much effort it would be. Eventually, the design would then look as follows:

Data
Main V8 Heap (most V8 objects)
Untrusted stack (inside the sandbox)
Compiler data (inside the sandbox)
Other data owned by V8 (inside the sandbox)
Trusted Stack
Chromium binary and library
Other Heaps (system heap, Blink heap, ...)

Code
JavaScript Code (interpreted or JIT-compiled)
WebAssembly Code
V8’s Runtime Code (e.g. the builtins)
V8’s Garbage Collector
V8’s Compilers
Embedder Code (all of Blink)

Consider again the stacktrace from above, but now with the final hardware-based sandbox design:

```
#0  v8::internal::JSArrayBuffer::Setup
#3  0x000055555af1b183 in v8::internal::Builtin_ArrayBufferConstructor
#4  0x000055555ec1ad7d in Builtins_CEntry_Return1_ArgvOnStack_BuiltinExit ()
#5  0x000055555e89d05d in
Builtins_InterpreterPushArgsThenFastConstructFunction ()
#6  0x000055555f25321f in Builtins_ConstructHandler ()
#7  0x000055555e89bd0f in Builtins_InterpreterEntryTrampoline ()
#8  0x000055555e89bd0f in Builtins_InterpreterEntryTrampoline ()
#9  0x000055555e89bd0f in Builtins_InterpreterEntryTrampoline ()
#10 0x000055555e8928dc in Builtins_JSEntryTrampoline ()
#11 0x000055555e892607 in Builtins_JSEntry ()
#15 0x000055555adcdd7d in v8::Script::Run
#17 0x000055555ad2e42d in v8::Shell::ExecuteString
#21 0x000055555ad4af2d in v8::Shell::Main
#22 0x000055555ad4b522 in main
```

Here, sandboxed mode would be entered effectively immediately when entering V8 code, for example through any of the public APIs, and wouldn't be left until returning from V8 to the embedder.

Pros:

- At least in theory, it would be possible to sandbox (almost) all of V8.
- Even in the simplest form, the stack can be sandboxed, which would mitigate additional bugs compared to the hardware-assisted sandbox, such as stack corruption bugs that would otherwise likely lead to a sandbox escape.
- Would (eventually) require fewer transitions between trusted and untrusted code, thereby potentially being more performant than a hardware-assisted sandbox (somewhat dependent on the stack switching overhead, see next point).

Cons:

- Switching the stack when crossing the privilege boundary might be expensive.
- Replacing software mechanisms might make the sandbox *less* secure, at least in the short term. For example, if privileged code wrongly assumes that a full pointer read from inside the sandbox can be trusted, it might subsequently corrupt trusted memory, leading to a sandbox escape. In contrast, when no full pointers can be stored inside the sandbox, this cannot happen.
- Difficult to implement: for example, it's unclear how to deal with global variables that are written to from C++ code.

Summary

It appears that a hardware-assisted sandbox would be the best option, at least initially, due to the simpler implementation effort and lower risk of introducing new problems. From there, if desired, it would be possible to gradually move towards a hardware-based sandbox by moving more memory regions into the sandbox, and running (some of the) C++ code also in the sandboxed mode.

Finally, it should be noted that none of the design choices are expected to provide significant performance *gains*: the main performance cost of the sandbox comes due to the pointer table (in particular the external pointer table, containing amongst others the pointers to Embedder objects), which will not be affected by either of the designs. All other parts of the software-based sandbox are sufficiently cheap that replacing them would likely not have a noticeable impact.

Hardware Requirements

This section summarizes the hardware requirements that would be necessary to implement any of the previously discussed designs. These requirements are, for the most part, independent of the concrete design choices, such as whether to use a hardware-assisted or hardware-based sandbox, and the amount of code that would run in the sandboxed mode.

Necessary features:

1. **Pages or virtual address ranges must be "taggable".** When executing sandboxed code, access to privileged memory must be forbidden. As such, there must be a way to mark pages or address ranges as either "privileged" or "unprivileged".
2. **Mode switching must be *fast***. It is expected that crossing the privilege boundary will happen very frequently (especially in the hardware-assisted design): every call into Embedder-provided APIs (all web APIs in Blink), as well as many calls inside V8 itself, e.g. to the garbage collector or other parts of the runtime. Likely, a performance cost similar to PKEYs (WRPKRU can cost >100 cycles) would be significantly too large. On the other hand, every crossing of what would become the privilege boundary already incurs the cost of a function call today, so a cost in that order of magnitude may be acceptable.
3. **Revoking write- but not read access must be possible.** It is unlikely that the unprivileged code can become fully hermetic any time soon as it needs to read various bits of information from outside of the sandbox (e.g. from the Isolate object or other, trusted space objects, the binary's data section). As such, it must be possible to revoke write access while still allowing read access.

Optional features:

1. **Provide a way to change the default permissions.** While it would be possible to hook into the low-level page allocation routines (e.g. mmap on Linux) to mark all non-sandbox pages as privileged, such an approach is somewhat fragile, and so a mechanism to limit access to the "default" pages would be preferable.
2. **Allow for more than two "regions"**. In its simplest form, the sandbox would require separating the address space only into two different regions: the sandbox (and potentially the stack), and everything else. Privileged code can access all memory, sandboxed code can only write to sandboxed memory. However, it may be helpful to have additional regions. For example, it might allow having more than one sandbox per process (but even then, at most one sandbox would be active for a given thread at any time). Further, it might be interesting to uniquely mark the *code pages* of sandboxed code to apply further restrictions to it (see next point).
3. **Stronger sandboxing mode.** In addition to limiting memory permissions, the sandboxed mode could enable further restrictions. For example syscalls could be disallowed entirely, and control-flow transfers to other code could be prohibited. Without this, it might be necessary to implement lightweight assembly verification for runtime-generated code (which would need to happen in privileged mode).

Examples

PKEYs

Conceptually, a PKEY-based sandbox would be simple: the sandbox region (and only the sandbox region) would be tagged with key 1. All other memory uses a different key (e.g. key 0, the default). Then, on every transition into and out of the sandbox, a WRPKRU instruction would be executed to revoke or restore write access to the other PKEYs, in particular to the default key used for (most) non-sandbox memory. The main issue would likely be the performance of the WRPKRU instruction which, with potentially >100 cycles required, would likely be significantly too slow to be used in practice, at least for a hardware-assisted sandbox.

HFI

The mechanisms provided by hardware-fault isolation can be used to implement a hardware-assisted or hardware-based sandbox. A possible configuration would specify the sandbox as one implicit memory region and the pages containing sandboxed code as an implicit code region. When the sandbox is entered, write access to all other memory is revoked,

and control-flow transfers to non-sandbox code would be forbidden. Further, the code would also be prohibited from performing any syscalls.

In case of a hardware-assisted sandbox where the stack must be accessible, the stack could be located in an explicit data region, in which case V8's compiler would need to emit special instructions for accessing stack memory.

As the sandbox does not attempt to prevent read access to privileged memory, protection from side-channel attacks such as Spectre are not necessary. The sandbox can therefore use the (significantly cheaper) non-serializing enter/leave operations.

All in all, HFI appears to fulfill all requirements for hardware-supported sandboxing. Whether the performance impact is acceptable in practice (especially for a hardware-assisted sandbox, which would be the preferred initial use-case) would still need to be determined experimentally.