V8 Sandbox - Trusted Space

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

This document is part of the V8 Sandbox Project and discusses the design of a new trusted heap space used to move V8 HeapObjects (i.e. objects managed by the garbage collector) out of the sandbox and reference them in a memory-safe way. For a general overview of the sandbox design see the high-level design document.

Background

There are certain (internal) HeapObjects used by V8 that, if manipulated by an attacker, can lead to memory corruption outside of the V8 sandbox, and thereby allow an attacker to bypass the sandbox. Examples of such objects include `BytecodeArrays` or `DeoptimizationData`, which would both allow an attacker to corrupt stack memory (and thereby escape from the V8 sandbox) if manipulated. More generally, objects that either contain code (machine code or interpreted bytecode) or code metadata can frequently cause issues if modified by an attacker. Throughout this document, such objects will be referred to as “trusted objects” as some part of V8 (implicitly) trusts that they have not been modified in a malicious way.

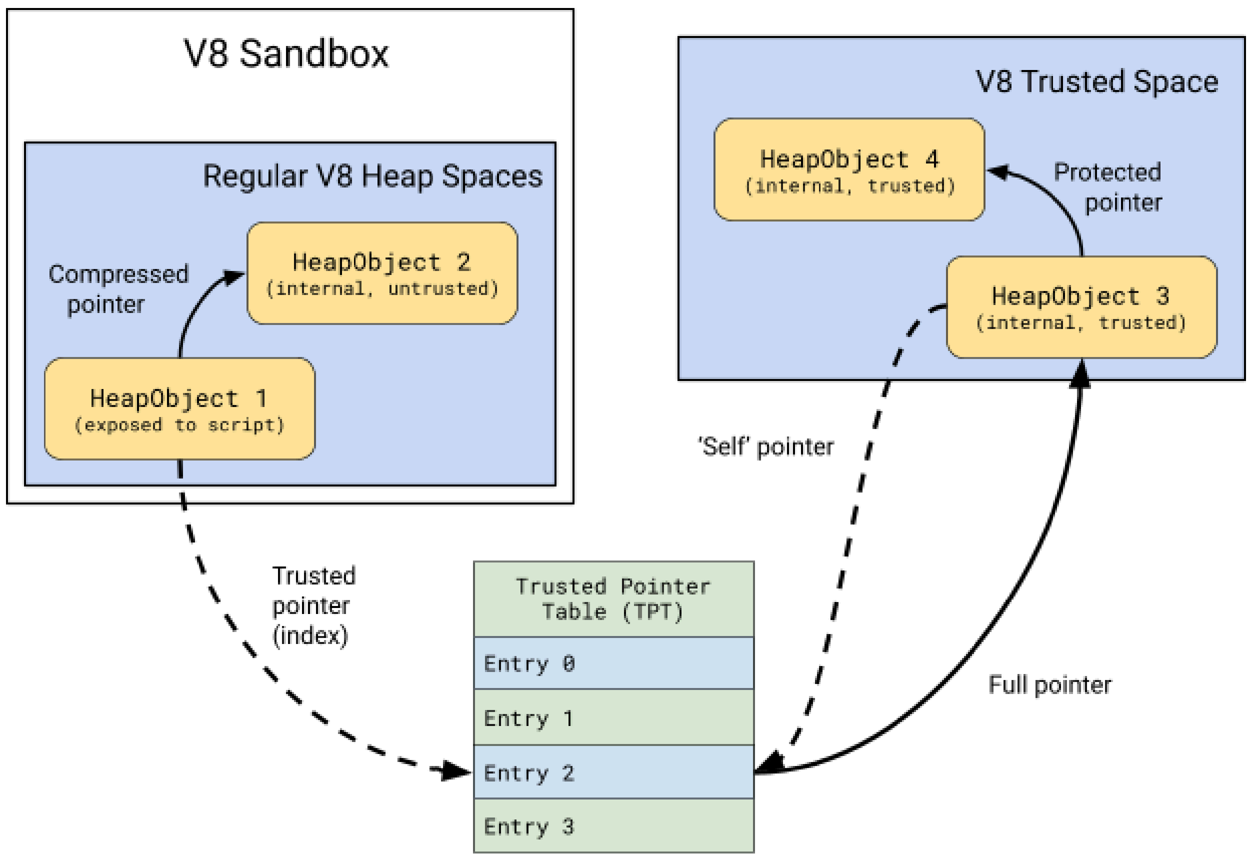
Objective

Prevent an attacker from abusing trusted objects to escape from the V8 sandbox.

Design

The basic idea behind this design is to simply move trusted objects out of the V8 sandbox. This way, attackers cannot directly manipulate these objects as it is assumed that they can only corrupt memory inside the sandbox.

For that, this design uses two building blocks: (1) a new trusted V8 heap space located outside of the sandbox and (2) a “trusted pointer” mechanism for safely referencing objects in that space from within the sandbox. The following diagram summarizes the design:



Here, a “trusted pointer” is implemented as an index into the trusted pointer table (TPT), which then contains a full pointer to a HeapObject living in the trusted space. A “protected pointer” on the other hand is simply a compressed pointer inside trusted space. For these pointers, it can be assumed that neither the pointer itself, nor the pointed-to object can be manipulated by an attacker, hence they are “protected”.

This design has a number of advantages:

- No (major) changes to the garbage collector are necessary: the trusted space will be managed in the same way as the other spaces, just its location in memory will be different.
- Support for compacting GC is possible without the need to introduce a new remembered set type for trusted pointers: instead, whenever a trusted object is moved around during compacting GC, it simply updates the raw pointer to itself through its ‘self’ pointer. This way, all trusted pointers from inside the sandbox stay valid.
- Type-safe access to trusted objects is possible for example by encoding the 16-bit instance type into the top bits of the pointer in the trusted pointer table. Alternatively, if there is only a small number of trusted object types, one could also use a dedicated pointer table per type.

- Thread-safe access to trusted objects is possible by using a per-Isolate trusted pointer table.

In the following, the two building blocks are discussed in more detail.

The Trusted Space

To be able to allocate objects outside of the sandbox, the following changes are necessary:

1. A new pointer compression cage must be allocated outside of the sandbox. Inside this cage, objects can reference each other through 32-bit compressed pointers. This way, references between objects in the trusted space, for example from a `Code` object to its `DeoptimizationData`, do not require a pointer table indirection.
2. A new "trusted" heap space, similar to the old space, must be introduced. It will be necessary to create both a regular trusted space and a large-object trusted space.
3. A pair of trusted spaces must be created inside the new pointer compression cage for every Isolate, since these objects are directly owned by an Isolate.
4. A trusted read-only space must be created inside the new pointer compression cage so that trusted read-only objects, such as builtin `Code` objects and the `Map` objects for trusted objects, are supported.

Apart from being located in a different pointer compression cage, the trusted space is otherwise very similar to the existing spaces and uses the same memory layout and garbage collection algorithms.

Trusted Pointers

Objects in the trusted space cannot be referenced through direct/raw pointers from within the sandbox as an attacker could manipulate these pointers and thereby effectively achieve memory corruption inside the trusted space. Instead, they need to be referenced through a pointer table indirection that guarantees memory-safe access, similar to the external pointer table.

When the sandbox is disabled at build time, these references must still function, but do not need to use a pointer table indirection. As such, we define a "trusted pointer" to be a reference to a trusted object that is implemented as either a regular tagged pointer when the sandbox is disabled, or an index into the trusted pointer table (an "indirect pointer") when the sandbox is enabled. Indirect pointers are then only available when the sandbox is enabled, while trusted pointers exist in all build configurations.

Adding support for trusted- and indirect pointers to V8 requires at least the following changes:

1. Adding a new pointer table: the trusted pointer table (TPT). It is essentially the generalized form of the code pointer table (CPT), containing not just pointers to Code

objects but instead to any kind of trusted HeapObject. To guarantee type-safe access, the entries in the table are tagged with the instance type of the pointed-to object.

2. Adding low-level trusted pointer accessor functions (`ReadTrustedPointerField` and `WriteTrustedPointerField`). These need to be added to at least the C++ runtime code as well as to the code stub assembler (CSA), the optimizing compilers, and the low-level macro assemblers.
3. Adding a new object visitor method (`VisitIndirectPointer`) which is used by the garbage collector to mark indirectly-referenced objects as alive. This visitor method is further used by the snapshot serializer as it needs to be able to serialize these kinds of references as well.
4. Adding new kinds of write barriers: since trusted objects are still tracked by the garbage collector, it must, during incremental GC, be informed whenever a field in an object that was already visited is changed to point to a different object. Write barriers accomplish this task. Since existing write barriers only know about direct/tagged pointer fields, a new kind of write barrier for indirect pointer fields needs to be added to the compilers, assemblers, and to the runtime.

While the read accessor for trusted pointers is fairly straightforward:

```
TrustedObject
HeapObject::ReadTrustedPointerField(int offset,
                                     InstanceType expected_type) {
#ifdef V8_ENABLE_SANDBOX
    uint32_t handle = ReadField<uint32_t>(offset);
    Address encoded_ptr =
isolate->trusted_pointer_table().Get(handle);
    Address decoded_ptr = encoded_ptr ^ (expected_type << 48);
    return HeapObject::cast(Object(decoded_ptr));
#else
    return ReadTaggedPointerField<TrustedObject>(offset);
#endif
}
```

It is not obvious how the write accessors should work. One option is that they simply copy the ‘self’ indirect pointer from the pointed-to object, and that all objects that live in the trusted heap inherit from the same base class that contains this value at a fixed offset. Then the accessor would simply be:

```
void
```

```
HeapObject::WriteTrustedPointerField(int offset, TrustedObject value)
{
#ifdef V8_ENABLE_SANDBOX
    uint32_t handle =

value.ReadField<uint32_t>(TrustedObject::kSelfIndirectPointerOffset);
    WriteField<uint32_t>(offset, handle);
#else
    WriteTaggedPointerField<TrustedObject>(offset, value);
#endif
}
```

Security Implications

Moving trusted objects out of the sandbox does not change the fact that they are security critical - it just prevents an attacker from directly corrupting them. As such, special care must be taken when modifying trusted objects, as any accidental memory corruption would likely result in a sandbox bypass. In general, any code that modifies data inside the trusted space must be considered sandbox attack surface.

To keep the sandbox attack surface small and manageable, the number of trusted heap objects should therefore be kept to a minimum, and they should be modified as rarely as possible (ideally, the objects are read-only after creation). Further, it is likely desirable that JIT-generated code cannot operate on these objects at all, as JIT-generated code might be influenced by (attacker-controlled) heap state.

Finally, objects directly accessible by script (such as strings and objects extending `JSReceiver`) should *never* be located in the trusted space as that might allow them to be corrupted. For example, if an attacker can construct a type confusion between an object in the trusted space and a “regular” object, this might allow the attacker to corrupt the trusted object. This property can be enforced to some degree by enforcing that all trusted objects extend a specific base class (e.g. `TrustedObject`) that is not compatible with `JSReceiver`, and so cannot directly be exposed to script. Furthermore, the fact that these objects live in a separate pointer compression cage should also make it harder (although not impossible) to accidentally “leak” them to JavaScript code as they aren’t directly addressable from inside the sandbox.

Alternatives Considered

Reusing the external code space

Instead of creating a new trusted space, it would also be possible to reuse the existing external code space and allocate the objects there. However, this would conflict with CFI, since the external code space must be executable (as it contains the JIT-generated machine code), while the trusted objects should not live in executable memory. While it would be possible to partition the external code space into an executable and a non-executable region, this might cause fragmentation of the code space. Further, the code space is supposed to be located close to the binaries .text section so calls to builtin functions can be short jumps/calls, while this is not required for trusted objects. As such, it is likely better to create a separate trusted space for (data) objects.

Turning HeapObjects into regular C++ objects

Instead of creating a trusted heap and moving HeapObjects there, it would also be possible to convert these HeapObjects to "regular" C++ objects (allocated through malloc in the system heap). Then they would be treated like other trusted objects and be referenced through the external pointer table. While this might be relatively easy for "leaf" objects such as the handler table `ByteArray` associated with a `BytecodeArray`, it becomes more difficult for objects such as `Code`, which themselves reference other objects. Besides requiring GC awareness for references between these C++ objects (or alternatively, switching them to manual memory management), it would also require adjusting the GCs heuristics for memory pressure. As this approach does not seem to have overall security or performance benefits over a trusted space, the latter then looks like the better option.

Protecting objects through memory permissions

In theory, it would be possible to protect trusted objects by allocating them in a read-only region inside the sandbox. However, in practice, this is problematic for a number of reasons:

1. It would require switching the permissions of the read-only region whenever an object is modified, which is likely too expensive, even with "fast" permission switching such as Intel PKEYs (which are not yet widely available). If per-process memory permissions are used, this would also leave the mechanism vulnerable to races as other threads controlled by an attacker might write into the read-only region while it is writable.
2. There would have to be a separate mechanism to guarantee that an object that is supposed to live in read-only memory actually does live there. As an example, consider the case where `BytecodeArrays` are allocated in read-only memory and are referenced from a `JSFunction` object through a regular compressed pointer. In this scenario, an attacker could simply craft their own fake `BytecodeArray` inside the regular (writable)

part of the heap, then make the `JSFunction` point to this object instead, which would be unsafe.

However, it is possible to keep objects in read-only memory inside the sandbox if the following conditions are met:

- These objects are never written to after initialization so that memory permissions do not need to be changed
- These objects are also referenced through indirect pointers from the (writable) heap so that the attack described above is not possible

This could for example be the case for builtin `Code` objects which are placed in the read-only heap.

Making all objects robust against corruption

Instead of moving or protecting objects, these objects could also be redesigned so that they are robust against manipulation by an attacker. However, in practice this would for example mean redesigning the interpreter (Ignition) and its bytecode so that corrupted bytecode can never result in memory corruption outside of the sandbox (in particular, on the stack). Alternatively, it would require inserting a large number of CHECKs into all code that operates on these objects, which is both expensive and fragile. As such, this approach is likely not generally applicable.

However, while it is unlikely that all objects can be made robust against manipulation by an attacker, it might be possible to harden some such objects and thereby avoid having to move them out into the trusted space.

Note that from a security point of view, it is not always obvious which of the two approaches is better: if objects are kept inside the sandbox, then the code that *reads/consumes* them becomes the attack surface (e.g. the bytecode interpreter). If they are moved out of the sandbox, then the code that *writes/produces* to them becomes the attack surface (as it can potentially be tricked to corrupt the object). As such, this decision likely needs to be made on a case-by-case basis.