V8 CFI Plans

Last update: 05/2022
Status: implementation in progress
Visibility: SHARED EXTERNALLY

Outline

This document describes a proposal to add control-flow integrity to V8. Our goal is to prevent arbitrary shellcode execution.

Besides the common forward- and backward-edge CFI, we need to solve protecting the output of the JIT compiler.
We're planning to use a **verifier** that checks the integrity of the assembler output before making it executable. Integrity in this case means that only a subset of the instructions can be emitted (e.g. no syscall instruction) and that direct and indirect jumps end up at allowed targets.
We assume that the attacker has a concurrent arbitrary read/write primitive. Since the verifier is running in the same process, it needs to protect its data using per-thread memory permissions.

Hardware requirements:

- Backward-edge CFI: ideally a shadow stack (CET) or separate control stack, alternatively signed return addresses (PAC)
- Per-thread memory permissions with fast switching
    - E.g. memory protection keys (PKU/PKEYS) or APRR (will use PKEY as the example going forward)

OS requirements:

- PKEY support in the OS
- Memory sealing: prevent unmapping pkey-protected memory (details)
    - This could be avoided if the key is part of the mapping/address.
- PKEY-protected signal/exception handlers (details)

The rest of the document will discuss the challenges we're aware of, i.e. how an attacker can gain native code execution, and how we're planning to address them.

Design

Return address corruption

Attackers can overwrite return addresses on the stack and do ROP.

Mitigation:
We're planning to rely on hardware features to protect against these attacks:

- Intel CET (shadow stack)
- PAC on ARM

Forward-edge protection

Attack can overwrite function pointers or vtables to gain *$pc* control.

Mitigation:
We will use forward-edge CFI to limit valid targets for indirect calls. We will start with a coarse-grained CFI scheme, i.e. we will allow all address-taken functions as targets and not perform function signature checks. The reasoning behind this is:

- Coarse-grained CFI will have lower performance overhead.
- In our model, the attacker can overwrite any function pointer. It's possible that they will find one with the right signature.
- Since the attacker is concurrent, they can overwrite the stack from other running threads. Instead of calling the gadget, they can wait for another thread to execute the target function and mess with local variables.

While fine-grained CFI will impose stricter constraints, it's not clear yet if it's worth the extra performance overhead and we will reassess this at a later point in time.

Note that we have to update valid call targets at runtime for JIT compiled code.

The particular CFI scheme to use is hardware and OS dependent:

- Windows: system DLLs already use CFG and it will be the easiest for compatibility to adopt it as well.
    - When allowlisting JIT compiled code as valid function entry points, we need to ensure that the attacker can't mess with the data. For that, we need to make sure to only perform this operation while our thread's memory is protected with pkeys. (more details later)
- Other OSes: everywhere else, we can use hardware support like Intel CET's ENDBRANCH or ARM BTI. With these features, any indirect call needs to end at an endbranch instruction, i.e. valid targets need to be specifically marked as such. There are some extra challenges with this approach:
    - We need to ensure that JIT code doesn't include (unaligned) endbranch instructions. This means we will have to mask constants in code to prevent emitting this byte pattern.

    - System libraries need to be compiled with endbranch support. Though we explicitly don't want all function entry points marked with endbranch instructions, only address-taken functions. We will need to detect at runtime if the system libraries have endbranch support.

JIT compilation

The attacker can overwrite the machine code generated by the JIT compiler. This doesn't just include the final code, but also intermediate buffers and local variables used by the assembler if they're stored on the stack.

Mitigation:

We can ensure that the generated machine code has certain properties by building a verifier that checks what kind of instructions the code will execute. The verifier itself needs to run in a way that the attacker can't overwrite its data: the stack and any data it reads need to be protected with per-thread memory permissions (i.e. pkeys).

This could look roughly like:

- Enter secure state (enable pkey, change stack)
- Copy code to protected memory
- Verify code
- Copy code to executable region
- Leave secure state

Before we talk about the verification steps, there's a bunch of data we need to protect.

- pkey protected stack:
    - Create a stack at process initialization time, pkey protect it and store it in a global variable. That variable needs to be pkey protected too. We can do this by putting it into a page aligned struct and protecting it at startup.
    - Since we can have multiple threads trying to run the verification at the same time, we need to have a mutex for the stack pointer and can map a new stack after switching to keep the lock for as short as possible. Alternatively, we can keep more than one secure stack to reduce contention.
    - Note that we don't want to allocate a new stack without switching to another secure stack first, since we can't guarantee that the allocation doesn't spill the new pointer onto the stack. This could be solved on Linux by doing raw syscalls ourselves.
- Global variables:
    - We need to assume all global variables are untrusted. If we need to access some, we need to protect them during process initialization time as above. This includes for example the pkey id we're using for protected memory.
- Dynamic memory:
    - Any data we dynamically allocate needs to be pkey protected. For example, the mapping in which we copy the JIT code before doing our verification. We can write a std::allocator that only hands out pkey protected memory.
- Code space metadata:

    - When copying the verified code into the executable region, we need to know that the target address is a valid code space address (we don't want to overwrite our stack for example) and that there's no existing code that we might partially overwrite.
    - Thus, we need to track all code mappings and free/used memory. We already have to perform all writes to the code space from a pkey-protected thread. In addition to that, we need to perform all code space allocations from a pkey-protected thread too and keep the metadata somewhere secure.

What does the verification need to do?

- Only a subset of machine instructions can be emitted (e.g. no syscalls)
- All direct branches need to go to aligned instructions. For this, we need to track valid instruction starts for this code object (or take them as input from the assembler).
- All direct branches need to go to either the same code object, a valid entry of another code object or a runtime/builtin function.
- If we use indirect branches in the code, we need to verify that they conform with our CFI scheme. In particular, with CFG you need to call a verification function first.
- For CFI schemes that use endbranch instructions, we need to check that these byte patterns don't exist aligned or unaligned in the code. To prevent overlaps with the next code block, we can pad the code with any byte not in the pattern.

Finally, we need to allow this new function in our CFI scheme, e.g. by emitting an endbranch instruction at the function start or by calling the right system library function to allow it with CFG.

Variant

If the validation turns out to have too much performance impact, we can also experiment with executing the assembler or the whole compiler in the protected state. The downside is that there will be more data to protect and we expose a bigger attack surface.

Code Space Compaction

The garbage collector moves code objects around to prevent fragmentation. To do this, it needs to do two dangerous things:

- Write to the code space
- Perform relocations on the code object and all code objects referencing it

An attacker could mess with the data of the GC thread and make it corrupt the code space.

Mitigation:

We can again run the critical section of this code in a pkey-protected state and ensure that all data we need to trust is also pkey-protected. From the previous section, we already need to track where code objects live. This will allow us to avoid overwriting existing code and performing the relocations in a secure way.

In addition, we need to make sure to invalidate the CFG entry for the old function.

Signal handlers / Exception handlers

Signal handlers / exception handlers push the CPUs register state into user memory (e.g. the stack) and load it again when returning from it. A concurrent attacker could overwrite the register state and gain code execution by overwriting the *$pc* value or changing the state of the pkey register and thereby bypassing our mitigations.

Mitigation:

On Linux, we can use sigaltstack + pkeys to protect this data. At program start, we set up a sigaltstack that is pkey protected. The signal handler can then enable the pkey as its first instruction before handling the signal. We need to be careful with all data accesses inside the signal handler though since any non-stack data can be attacker controlled and lead to memory corruption.

We currently don't have an idea how to solve this on Windows. Maybe we could ask for OS help, e.g. sigaltstack support.

PC Fixups

We have cases in which v8 needs to update the *$pc* to point to a new location:

- During optimization/deoptimization we need to switch control flow between interpreter and optimized code.
- The wasm trap handler updates the *$p*c inside a signal handler.

Mitigation:

In both cases, we can verify the new *$pc* values at creation time and store them in pkey-protected memory. For example, when verifying generated wasm machine code, we can remember the fixup values and store them in pkey-protected metadata.

Similarly, when verifying the optimized JavaScript code, we can store the valid entry points in pkey-protected metadata.

Corrupt data of running threads

As mentioned before, we assume the attacker can overwrite any data from concurrent threads. In particular, there are many code paths that can get executed where it can lead to bad results if the attacker can overwrite local variables on the stack.

For example, imagine the garbage collector wants to unmap some page on the heap and the attacker can replace the address with the address of the pkey-protected stack. The kernel would then unmap the stack and can re-use the memory on the next mmap call, effectively clearing the pkey protection.

Mitigation:

Protecting against these attacks will need to focus on the worst cases we can find. E.g. to protect against the munmap example above, we could ask OS vendors to give us a way to seal these mappings or only allow unmapping them if the current thread has pkey write permissions to it.

This could also be avoided if the pkey was part of the address or if we can disable read/write access to the default key in our verifier.
Another bad example would be if the syscall number of a syscall is ever in attacker-writable memory. We'll need to find such cases and handle them one-by-one.

Command-line Flags

Security features can be disabled with command-line flags. An attacker can overwrite these flags in memory.

Mitigation:
We need to make the security critical flags read-only. To prevent the munmap attack, we can mark the page with a pkey and rely on the solution of the previous section.
Alternatively, we could JIT compile flags into small functions that return the value.

Kernel callbacks

Some syscalls have function pointers as arguments that are later used as callbacks. Signal handlers are one example for this. If the attacker can reach the signal handler registration and replace the function pointer argument, they can gain *$pc* control.

Mitigation:
This is an actual problem in the v8 code since the signal handler registration is guarded by a global variable and can be reached via a virtual function call. We need to look for similar cases and prevent this pattern through code changes.

Hardware Support

x64

- Intel Tiger Lake and later support PKU and CET (shadow stacks + landing pads).
- AMD Zen3 supports PKU and shadow stacks, but afaik no landing pads.

ARM

- Pointer Authentication
- Permission indirection and overlays
- Guarded Control Stack

Chromebook Hardware support