Memory access due to code generation flaw in Cranelift module

Critical **cfallin** published **GHSA-hpqh-2wqx-7qp5** on May 21, 2021

Package
cranelift-codegen (crates.io)

Affected versions	Patched versions
<= 0.73.0	0.73.1, 0.74.0

Description

There is a bug in 0.73.0 of the Cranelift x64 backend that can create a scenario that could result in a potential sandbox escape in a WebAssembly module. Users of versions 0.73.0 of Cranelift should upgrade to either 0.73.1 or 0.74 to remediate this vulnerability. Users of Cranelift prior to 0.73.0 should update to 0.73.1 or 0.74 if they were not using the old default backend.

Description

This bug was introduced in the new backend on 2020-09-08 and first included in a release on 2020-09-30, but the new backend was not the default prior to 0.73.0. The recently-released version 0.73.0 with default settings, and prior versions with an explicit build flag to select the new backend, are vulnerable. The bug in question performs a sign-extend instead of a zero-extend on a value loaded from the stack, under a specific set of circumstances. If those circumstances occur, the bug could allow access to memory addresses up to 2GiB before the start of the heap allocated for the WebAssembly module.

If the heap bound is larger than 2GiB, then it would be possible to read memory from a computable range dependent on the size of the heap's bound.

The impact of this bug is highly dependent on heap implementation; specifically:

- if the heap has bounds checks, and
- does not rely exclusively on guard pages, and
- the heap bound is 2GiB or smaller

then this bug cannot be used to reach memory from another WebAssembly module heap.

The impact of the vulnerability is mitigated if there is no memory mapped in the range accessible using this bug, for example, if there is a 2 GiB guard region before the WebAssembly module heap.

The bug in question performs a sign-extend instead of a zero-extend on a value loaded from the stack when the register allocator reloads a spilled integer value narrower than 64 bits. This interacts poorly with another optimization: the instruction selector elides a 32-to-64-bit zero-extend operator when we know that an instruction producing a 32-bit value actually zeros the upper 32 bits of its destination register. Hence, we rely on these zeroed bits, but the type of the value is still i32, and the spill/reload reconstitutes those bits as the sign extension of the i32's MSB.

The issue would thus occur when:

- An i32 value is greater than or equal to 0x8000_0000;
- The value is spilled and reloaded by the register allocator due to high register pressure in the program between the value's definition and its use;
- The value is produced by an instruction that we know to be "special" in that it zeroes the upper 32 bits of its destination: add, sub, mul, and, or;
- The value is then zero-extended to 64 bits;
- The resulting 64-bit value is used.

Under these circumstances there is a potential sandbox escape when the i32 value is a pointer. The usual code emitted for heap accesses zero-extends the WebAssembly heap address, adds it to a 64-bit heap base, and accesses the resulting address. If the zero-extend becomes a sign-extend, the module could reach backward and access memory up to 2GiB before the start of its heap.

This bug was identified by developers at Fastly following a report from Javier Cabrera Arteaga, KTH Royal Institute of Technology, with support from project Trustful of Stiftelsen för Strategisk Forskning. In addition to supporting the analysis and remediation of this vulnerability, Fastly will publish a related Fastly Security Advisory at https://www.fastly.com/security-advisories.

In addition to assessing the nature of the code generation bug in Cranelift, we have also determined that under specific circumstances, both Lucet and Wasmtime using this version of Cranelift may be exploitable.

General Impact to Lucet

Lucet inherits the heap address computation and bounds-checks of Cranelift, which it uses as its backend code generator. Of particular importance specifically is the address-space layout used by Lucet. In the default configuration for Lucet, only a single module is running, and therefore it is not possible to access memory from another module.

By default, the open source implementation of Lucet uses a maximum heap size of 4 GiB, and an instance slot size of 8 GiB, when invoking an instance from the lucet-wasi command-line tool. These settings are within the range of vulnerability described above, but only a single instance is running, so there is no other instance to read. When embedding the runtime (for example, in a long-running daemon), the default for the heap size as described in the source is 1MB; with this setting, the runtime is not vulnerable.

Lucet allocates its WebAssembly module instances into "instance slots", which are contiguous zones of virtual address space that contain the VM context at the bottom, the WebAssembly heap in the next page after that, a guard region in the middle, and other data at the top: the stack and the globals.

If the instance slot size is less than (max heap) + 2GiB, then the lowest accessible address using the bug will overlap with the prior instance's heap. If the size of VM context + stack + globals is greater than (4GiB - heap limit), then the highest accessible address using the bug will overlap with this critical data. If neither of these conditions are true, the bug should only result in an access to the prior instance's guard region.

Generally, if the limit is between 2GiB and 4GiB - ~1MB (depending on stack/global size) and the instance slot size is less than 6GiB, the configuration is vulnerable. If the limit is greater than 4GiB - ~1MB, the configuration is vulnerable regardless of instance slot size. Otherwise, the configuration is not vulnerable.

General Impact on Wasmtime

In Wasmtime, the same Cranelift heap address computations and heap types are used as above. The memory layout, however, is slightly different, with different outcomes:

- With the mmap implementation impact is mitigated probabilistically if ASLR is enabled.
- With the pooling allocator, the vulnerability only exists if a memory reservation size lower than the default of 6GB is used.

With the default mmap-based instance memory implementation, Wasmtime uses mmap() to allocate a block of memory large enough for the heap and guard region, as specified in its configuration. If the underlying OS implements ASLR (modern Linux, macOS and Windows do) then this address will be randomized, and the region below it will (probabilistically) be free. Hence, the bug is mitigated probabilistically in the default configuration if ASLR is enabled.

If using the pooling allocator, the vulnerability exists if instance memory size (`memory_reservation_size` in InstanceLimit) is strictly less than 6GiB (4 GiB + 2 GiB of guard pages). The default is 6GiB, so the vulnerability is masked in the default pooling allocator configuration.

Severity

Critical

CVE ID

CVE-2021-32629

Weaknesses

CWE-788