The AArch64 processor (aka arm64), part 18: Return address protection

devblogs.microsoft.com/oldnewthing/20220819-00

August 19, 2022

Raymond Chen

The ARMv8.3-A version of the instruction set introduces some new instructions that make it harder for attackers to modify return addresses on the stack. It is a special case of a feature called *pointer authentication*.

Even though pointers on AArch64 are 64-bit values, not all of the bits are used for addressing. The architecture supports using some of these bits in data pointers as "tag" bits (a feature Windows does not use, as far as I'm aware), and the remaining bits are required to contain canonical values: Zeroes for user-mode addresses, and ones for kernel-mode addresses.

Bits that must contain canonical values can therefore be reconstructed from the other bits of the address. This means that as long as we don't intend to use them to address memory, we can scramble those bits temporarily, as long as we restore the canonical values before using the pointer as an address.

For the purpose of this discussion, I will use the term "non-redundant" to refer to bits that are in positions not considered canonical. I am avoiding the term "non-canonical" because that could be interpreted as meaning "bits in canonical positions but which do not hold canonical values".

The first half of pointer authentication is scrambling the bits. There are many variations of this instruction, but this is the one that Windows uses:

```
; generate pointer authentication code
; for instruction address using key B
; and sp register
pacibsp
```

This instruction takes as input three values:

- non-redundant bits of *lr* register,
- 128-bit value in secret register "instruction-B",

- 64-bit value in *sp* register.

It hashes all of these values together and uses the result to replace the redundant canonical bits in the *lr* register.[1]

The reverse instruction is

```
; authenticate and restore instruction address
; that was encoded with key B and sp register
autibsp
```

This instruction takes the same inputs as the `pacibsp` instruction, runs them through the same hash function, and verifies that the redundant bits in the *lr* register still match the hash. If so, then it restores the canonical bits, thereby recovering the original value in *lr*. If not, then it puts intentionally invalid values in the canonical bits, so that attempting to return to *lr* immediately takes an access violation.

The idea is that you use `pacibsp` to hash the return address and the stack pointer (and some invisible state) before you store the return address to memory. After you load the return address from memory, you use `autibsp` with the same stack pointer (and the same invisible state) to confirm that the value has not been tampered with.

This pair of instructions makes it harder to carry out an exploit that involves overwriting the return address or building a return-oriented programming attack: The attacker must not only obtain a return address (already made harder due to address space layout randomization), but they must also figure out the correct hash value to put in the redundant bits of the address so that the address will pass the `autibsp` check that occurs after the return address is loaded from memory.

The other flavors of the pointer authentication instructions all follow the same pattern. They just vary in letting you specify

- Whether you are encoding or decoding,
- The register that contains the pointer,
- Whether it should be interpeted as an instruction or data address,
- Which secret register to use,
- What to use for the 64-bit value to mix in (a register, *sp*, or *zr*).

There are five secret registers:

- `IA` = “instruction-A”
- `IB` = “instruction-B”
- `DA` = “data-A”
- `DB` = “data-B”

- `GA` = “generic-A”

The “instruction” versions are used for encoding and decoding instruction pointers. The “data” versions are used for encoding and decoding data pointers. The “generic” version calculates and returns the raw hash, and it’s up to you to do your own encoding and decoding. (For example, you might store the hash next to the pointer.)

These new instructions were placed in an existing block of instructions architecturally labeled `hint` : Processors are required to treat any unrecognized instructions in this block as if they were `nop` . That way, new code can start using these instructions without having to check for processor support: The older processors just ignore the instruction, so code that uses the instructions still run fine; they just don’t get any added protection.

There are no user mode instructions for reading the secret registers directly. The only thing you can do with the secret register is ask it to encode or decode a pointer.

There are also some new instructions for pointer authentication. For these, you *do* have to check for processor support. (They can’t go in the `hint` space, because they actually do things.)

```
; Stripping PAC bits and restoring canonical bits
xpaci   Xn   ; restore canonical bits to instruction address
xpaclri      ; restore canonical bits to instruction address in lr
xpacd   Xn   ; restore canonical bits to data address

; Combo instructions (also have "b" variants)
; Decoded address is discarded after use (not stored back)
retaa                       ; autiasp + ret
braa    Xn                  ; autia + br
blraa   Xn                  ; autia + blr
ldra    Xt, [Xn/sp, #imm]   ; autda + ldr, reach 8 × (-512..511)
```

The Microsoft Visual C++ compiler does not generate these instructions due to the lack of backward compatibility. Furthermore, I wouldn’t be surprised if the ABI requires that functions return with a traditional `ret` instruction.

Bonus chatter: Since return addresses are not valid across processes, Windows gives each process a different value in the secret register “B”, thereby making it even trickier to generate false return addresses from outside the process.[2]

Bonus bonus chatter: The Windows debugger understands that return addresses on the stack are encoded, and when it generates a stack trace, it restores the canonical values in the return addresses.

[1] If the canonical bits in the *lr* register were not set properly on entry, then it puts an intentionally corrupt hash in the redundant canonical bits on exit, ensuring that the subsequent `autibsp` will fail.

[2] By comparison, when you fork a process in Unix, the secret registers[3] are shared between the child and the parent, because the child still has return addresses on the stack created by the parent. The system has to wait until the "exec" to change the values in the secret registers.

[3] I don't know whether linux uses secret register "A" or secret register "B" for return address protection.

Raymond Chen

Follow