Effects of classic return address tricks on hardware-assisted return address protection

devblogs.microsoft.com/oldnewthing/20241016-00

October 16, 2024

The x86-32 architecture notoriously does not offer direct access to the instruction pointer, and a common trick is to use `call`/`pop` to read the instruction pointer.

```
    ; read current address into register
    call    @F
@@: pop     eax     ; eax = current address
```

And since x86-64 does not offer an absolute jump instruction, it is a common trick to use a `push`/`ret` as a substitute.

```
    ; jump to absolute address
    push    0x12345678
    ret             ; jump to 0x12345678
```

We learned a while back that these unmatched `call`/`ret` pairs unbalance the return address predictor¹ and end up being net pessimizations.

And we recently learned that they also unbalance the hardware shadow stack, and the consequences of that are even worse: Instead of merely damaging your performance, this code doesn't run *at all* because it also unbalances the hardware shadow stack, and an improper return results in an exception.

In the case of Windows, the kernel receives the exception and checks whether the code performing the invalid `ret` is marked as compatible with return address protection. If so, then any return address protection failure is considered fatal. If not, then the kernel tries to forgive the error by popping entries off the hardware shadow stack until it finds a return address that matches the one popped from the CPU stack. If no match is found, then the failure is treated as fatal.

If you do a `push`/`ret`, that return address you pushed is nowhere in the valid return address history, and the kernel will terminate the process.

If you do a `call`/`pop`, then you pushed an extra entry onto the shadow stack, and what happens next varies.

If your function ends with a `ret`, then that `ret` will be mismatched, and the kernel notices that it occurred inside a DLL that is marked as “not CET compatible”, so the kernel will shake its head, “oh man, here’s a weirdo”, and it will look up the stack and find the true return address one entry higher.

If your function ends with a tail call optimization that jumps to another function, then that other function’s `ret` will be the one that takes the mismatch exception. If that other function is in a DLL that is marked as “CET compatible”, then the kernel will say, “That’s a paddlin’” and terminate the process.

So the `push`/`ret` pattern results in a guaranteed process termination, whereas the `call`/`pop` might result in a process termination depending on how lucky you feel.

(Not recommended.)

¹ It appears that this specific pattern of `call`/`pop` is special-cased inside modern processors and does not unbalance the return address predictor stack after all.