A quick introduction to return address protection technologies

devblogs.microsoft.com/oldnewthing/20241015-00

October 15, 2024

Return Oriented Programming (ROP) is a malware technique that takes advantage of a memory write vulnerability to populate the stack with synthesized return addresses, each of which points to a code fragment (known as a *gadget*) that executes a few instructions before performing a return instruction. The idea is that an attacker can gain arbitrary code execution by cobbling together these small sequences of instructions into a larger operation.

A common defense against ROP techniques is to use some form of return address protection by confirming that the return address that is about to be used matches the return address received at the start of the function. In the case of a ROP, the synthesized return addresses do not correspond to any call, and this gives the system an opportunity to detect that something bad has happened.

We saw some time ago that the AArch64 architecture contains hardware support for return address validation through the use of the `pacibsp` and `autibsp` pair of instruction which sign a return address and validate the signature, respectively.

Another approach is to use a *shadow stack*, which is another stack in memory into which copies of the original return addresses are recorded, and against which those return addresses are validated before being used.

There are two common patterns for shadow stacks, known as *parallel shadow stacks* and *compact shadow stacks*.

The *compact shadow stack* reserves another register to be used as a shadow stack pointer. For example, you might do this:

```
; function entry with return address on CPU stack
; assume r15 is the shadow stack pointer

    ; retrieve return address
    mov     rax, [rsp]

    ; push onto shadow stack
    mov     [r15-8], rax
    lea     r15, [r15-8]

    ⟦ main function body goes here ⟧

    ; before returning, pop the return address
    ; from the shadow stack
    mov     r11, [r15]
    lea     r15, [r15+8]

    ; and check that it matches the CPU stack
    cmp     r11, [rsp]
    jnz     fatal

    ret
```

This is called a compact shadow stack because all the return addresses are stored in contiguous memory. The amount of memory required for the shadow stack is `sizeof(address)` × call depth.

CPU stack	shadow stack	
⋮	⋮	
retaddr1	retaddr1	
local var	retaddr2	
local var	retaddr3	← r15
retaddr2		
local var		
local var		
local var		
retaddr3		
local var		
local var	← rsp	

By comparison the *parallel shadow stack* allocates a block of memory the same size as the CPU stack, and there is a buddy system between each byte of the CPU stack and each byte of the shadow stack. Access to the shadow stack is usually mediated by an otherwise-unused selector.

```
; function entry with return address on CPU stack
; assume fs has a base address equal to the distance
; between the CPU stack and the shadow stack

    ; retrieve return address
    mov     rax, [rsp]

    ; copy to shadow stack
    mov     fs:[rsp], rax

    ⟦ main function body goes here ⟧

    ; before returning, compare the return address
    ; to the shadow stack
    mov     r11, fs:[rsp]
    cmp     r11, [rsp]
    jnz     fatal

    ret
```

This is called a parallel shadow stack because the two stacks run parallel to each other.

CPU stack		shadow stack
⋮		⋮
retaddr1		retaddr1
local var		
local var		
retaddr2		retaddr2
local var		
local var		
local var		
retaddr3		retaddr3
local var		
local var	← rsp	← fs:rsp

Here's a table of pros and cons:

	Compact	**Parallel**
Code size	Larger	Smaller
Memory consumption	Smaller	Larger
Register pressure	Greater	Smaller

Although both the compact and parallel stacks require a new dedicated register, the compact stack takes the register from the general purpose registers, which makes it unavailable for code generation. The parallel stack uses a selector that would otherwise go unused.

A significant problem with software-based return address protection on x86-64 is that the return address is passed from the caller to the callee via memory, which opens a race condition (page 29) where an attacker can modify the return address in memory after it has been pushed by the `call` instruction but before it is loaded by the `mov rax, [rsp]` at the start of the called function. (This is not a problem for processors which use a link register to pass the return address.)

Intel Control-flow Enforcement Technology (CET) implements a compact shadow stack in hardware using a dedicated register not visible to user mode. When active, `call` instructions automatically push return addresses on to the shadow stack, and `ret` instructions automatically pop and validate return addresses from the shadow stack. Performing the shadow store as part of the `call` instruction removes the race condition.

Okay, that was a lot of stuff just to provide the required reading in anticipation of the real topic, which we'll pick up next time.

Bonus chatter: Some versions of return address protection simply ignore the return address on the CPU stack and just use the value from the shadow stack. Corrupt the return address all you want; we don't use it!

```
; compact shadow stack version

    ; on function entry,
    ; push return address onto shadow stack
    mov     rax, [rsp]
    mov     [r15-8], rax
    lea     r15, [r15-8]

    ⟦ main function body goes here ⟧

    ; return to the address on the shadow stack
    pop     r11             ; discard CPU stack
    mov     r11, [r15]      ; fetch from shadow stack
    lea     r15, [r15+8]    ; pop from shadow stack
    jmp     r11             ; go to where the shadow stack tells us

; parallel shadow stack version

    ; on function entry,
    ; copy return address to shadow stack
    mov     rax, [rsp]
    mov     fs:[rsp], rax

    ⟦ main function body goes here ⟧

    ; return to the address on the shadow stack
    pop     r11             ; discard CPU stack
    mov     r11, fs:[rsp]   ; fetch from shadow stack
    jmp     r11             ; go to where the shadow stack tells us
```

You could go even further and remove the return address from the CPU stack entirely, which saves an instruction and also permits a more compact encoding.

```
; compact shadow stack version

    ; on function entry,
    ; pop return address from CPU stack
    ; and push to shadow stack
    pop     rax
    mov     [r15-8], rax
    lea     r15, [r15-8]

    ⟦ main function body goes here ⟧

    ; return to the address on the shadow stack
    mov     r11, [r15]      ; fetch from shadow stack
    lea     r15, [r15+8]    ; pop from shadow stack
    jmp     r11             ; go to where the shadow stack tells us
```

Exercise: Why can't we use the "transfer the return address to the shadow stack and remove it from the CPU stack" technique for parallel shadow stacks?

This technique has multiple downsides. One is that it makes building stack traces much harder since you have to consult the shadow stack to figure out who the caller is. And the `jmp` instruction at the end unbalances the return address predictor. And this technique does not play friendly with CET: The shadow stack just grows and grows because no `ret` instruction is ever executed. And finally, this technique is not compatible with the Windows x86-64 ABI, which requires that return addresses be on the CPU stack.

Answer to exercise: You might think you could transfer the return address to the parallel shadow stack like this:

```
; parallel shadow stack version

    ; on function entry,
    ; pop return address from CPU stack
    ; and copy to shadow stack
    pop     rax
    mov     fs:[rsp], rax

    ⟦ main function body goes here ⟧

    ; return to the address on the shadow stack
    mov     r11, fs:[rsp]   ; fetch from shadow stack
    jmp     r11             ; go to where the shadow stack tells us
```

However, this doesn't work because it would mean that if your function consumes no stack space, then any function you call will overwrite your shadow stack entry with their return address.

Bonus bonus chatter: Shadow stacks adds another reason why Windows insists on allocating thread and fiber stacks rather than letting programs provide their own stack memory: A program-provided stack doesn't have an associated shadow stack.

(We learned another reason some time ago: The Itanium's backing store stack.)