The AArch64 processor (aka arm64), part 21: Classic function prologues and epilogues

devblogs.microsoft.com/oldnewthing/20220824-00

August 24, 2022

Raymond Chen

Classic function prologues in Windows on AArch64 follow a common pattern. I'll present the full prologue, and then we'll take it apart instruction by instruction.

```
; return address protection
pacibsp

; saving registers
stp     fp, lr, [sp, #-0x30]!
stp     x19, x20, [sp, #0x10]
str     x21, [sp, #0x20]

; establishing frame chain
mov     fp, sp

; initializing GS cookie
bl      __security_push_cookie

; local variables and outbound parameters
sub     sp, sp, #0x80
```

The prologue breaks up into five sections, as marked off by comments above.

On entry to the function, we have this:

return address	
previous *fp*	← *fp* (frame chain)
⋮	
stack param	← *sp*

The first order of business is to apply return address protection.

Next, we save nonvolatile registers. We build the next stack frame by pushing *fp* and *lr* onto the stack into adjacent locations. The frame pointer and link register are always stored next to each other because that's what stack walking requires. The "push" onto the stack is done by using a pre-incrementing store,[1] so that the stack pointer is adjusted, and then the values written to the adjusted stack pointer.

Let's walk through that "push" again:

```
    stp     fp, lr, [sp, #-0x30]!
```

The effective address is *sp* − `0x30`, which is `0x30` bytes below the current stack pointer. At that location, we store the *fp* and *lr* registers, and then the effective address is written back to the base register *sp*.

return address	
previous *fp*	← *fp* (frame chain)
⋮	
stack param	
return address	
previous *fp*	← *sp*

The next instruction stores the *x19* and *x20* registers into the register save area we just created.

```
    stp     x19, x20, [sp, #0x10]
```

return address	
previous *fp*	← *fp* (frame chain)

⋮	
stack param	
saved *x20*	
saved *x19*	
return address	
previous *fp*	← *sp*

And the last instruction in the set saves the lone *x21* register.

```
    str     x21, [sp, #0x20]
```

return address	
previous *fp*	← *fp* (frame chain)
⋮	
stack param	
saved *x21*	
saved *x20*	
saved *x19*	
return address	
previous *fp*	← *sp*

The stack must remain 16-byte aligned, which means that space must be reserved in multiples of 2 registers. We have an odd number of registers to save, so one of the spaces we reserved for the register save area goes to waste. (In theory, the compiler could decide to use it to record a local variable, but in practice it doesn’t.)

The second part of the prologue re-establishes the frame chain.

```
    ; link this frame onto the frame chain
    mov     fp, sp
```

	return address	
▶	previous *fp*	
	⋮	
	stack param	
	saved *x21*	
	saved *x20*	
	saved *x19*	
	return address	
	previous *fp*	← *fp* (frame chain), sp

If the function contains a stack-based array, then the prologue inserts the GS cookie onto the stack so that a buffer overflow from the stack-based array is likely to corrupt the cookie before it gets to the saved return address.

```
    bl      __security_push_cookie
```

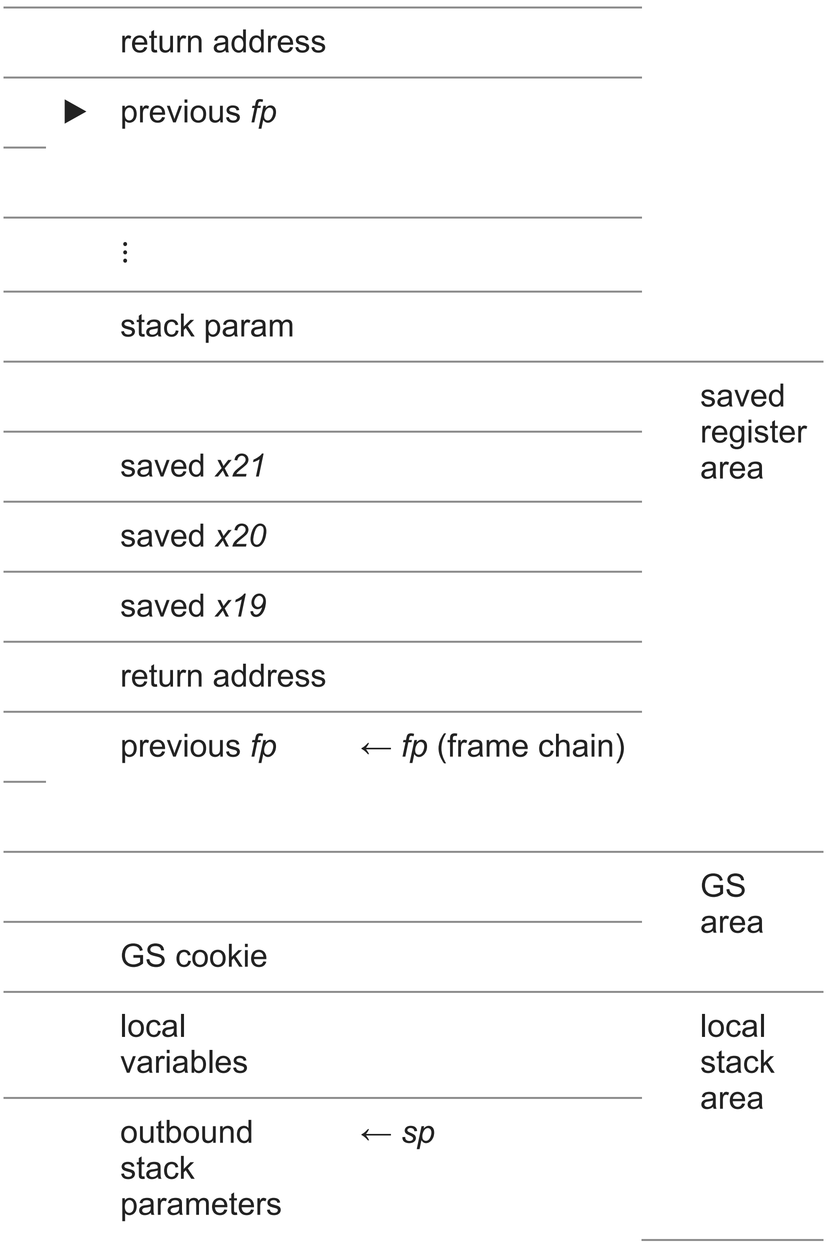
Even though the cookie is only the size of a register, the function pushes 16 bytes onto the stack due to the requirement that the stack remain 16-byte aligned.

The cookie management functions use the *xip0* and *xip1* registers to do the work of calculating or validating the cookie. These registers are volatile and are assumed to be modified by any branch instruction, and we used a branch instruction to get to the start of the prologue, so we know that *xip0* and *xip1* cannot be used to pass information from the caller to the callee, not even for a nonstandard calling convention.

Finally, the prologue allocates space for stack-based local variables and outbound parameters.

```
    sub     sp, sp, #0x80
```

We then reach the function body with this stack layout:



return address		
▶ previous *fp*		
⋮		
stack param		
		saved register area
saved *x21*		
saved *x20*		
saved *x19*		
return address		
previous *fp*	← *fp* (frame chain)	
		GS area
GS cookie		
local variables		local stack area
outbound stack parameters	← *sp*	

When the function returns, the above steps are reversed.

```
    add     sp, sp, #0x80           ; discard local stack area
    bl      __security_pop_cookie   ; validate and pop GS cookie
    ldr     x21, [sp, #0x20]        ; restore register
    ldp     x19, x20, [sp, #0x10]   ; restore registers
    ldp     fp, lr, [sp], #0x30     ; restore register and pop
    autibsp                         ; validate return address
    ret                             ; return
```

The final `ldp` uses the post-increment addressing mode so that the stack pointer is increased by `0x30` after the registers are loaded.

Restoring the *fp* register unlinks the current stack frame from the frame chain. And restoring the *lr* register puts the return address back into *lr*, which we validate, and then use in the `ret` instruction to return to the caller.[2]

Not all of these steps will be present in all function prologues. A function that has no stack-based local variables or outbound parameters will not create a local stack area. A function that has no stack-based arrays will not create a GS cookie. And a lightweight leaf function won't even bother saving any registers or protecting the return address.

Next time, we'll look at special cases that will diverge from this traditional prologue/epilogue pattern.

[1] Though since the offset is negative, you can think of it as a pre-*decrementing* store.

[2] In AArch64, the program counter *pc* is not a general-purpose register, so you don't see the trick popular in AArch32 where the return address is popped into the *pc* register to return to the caller. For AArch64, we see the more traditional pattern of restoring the return address into *lr* and then explicitly returning to it.

Raymond Chen

Follow