The AArch64 processor (aka arm64), part 22: Other kinds of classic prologues and epilogues

devblogs.microsoft.com/oldnewthing/20220825-00

August 25, 2022

Raymond Chen

Last time, we looked at traditional classic function prologues in Windows on AArch64. But there are variations.

For variadic functions, there is a small adjustment to the traditional prologue: The top of the register save area contains spill space for the variadic register parameters. The variadic register parameters are at the top so that they nestle directly against the stack-based parameters, so that the entire variadic parameter list can be accessed uniformly in memory.

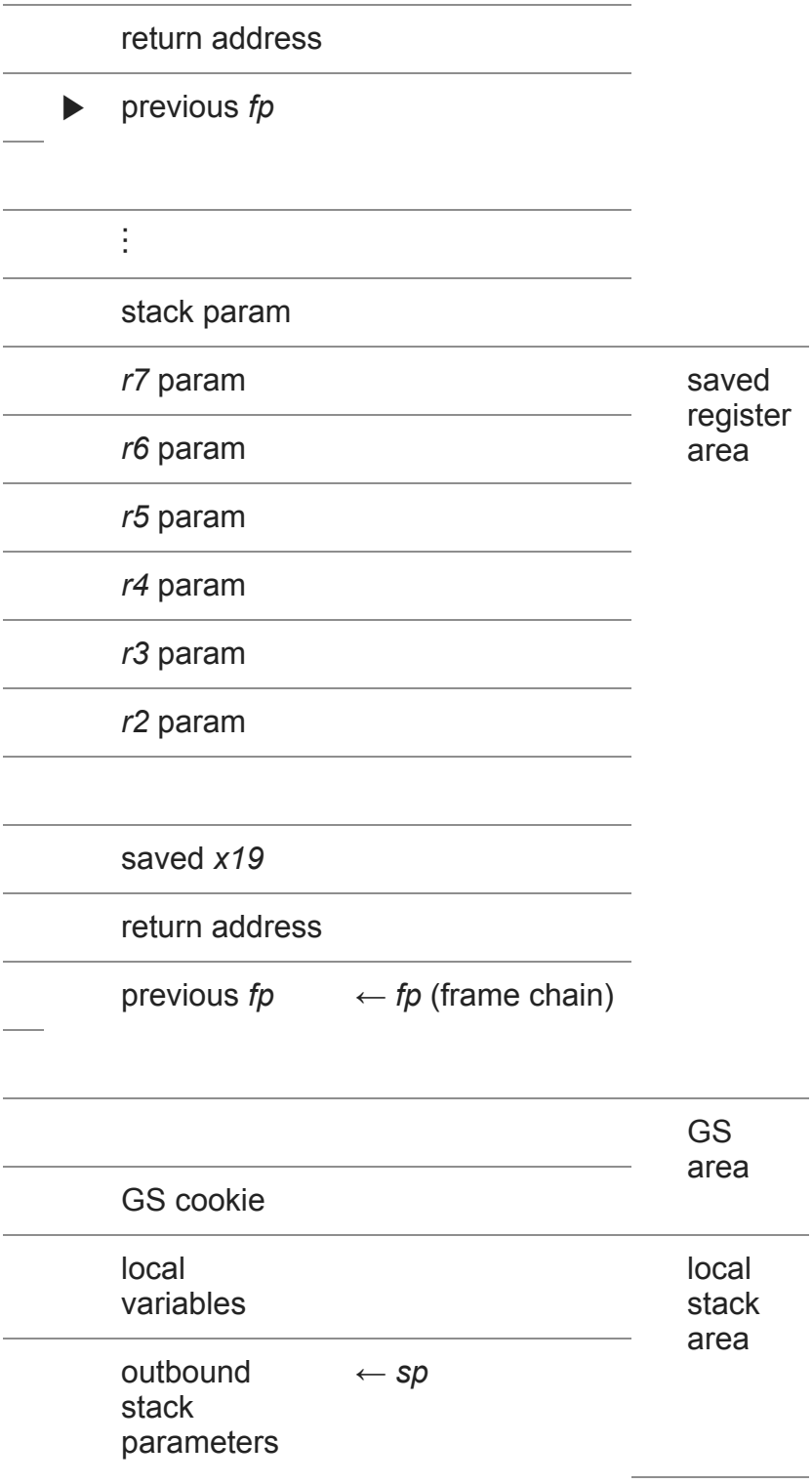
For example, a function whose prototype is

```
int something(int a, int b, ...);
```

could have a prologue that goes something like

```
    ; return address protection
    pacibsp

    ; saving registers + variadic parameters
    stp     fp, lr, [sp, #-0x50]!
    str     x19, [sp, #0x10]
    stp     x2, x3, [sp, #0x20]     ; variadic registers go here
    stp     x4, x5, [sp, #0x30]     ; variadic registers go here
    stp     x6, x7, [sp, #0x40]     ; variadic registers go here

    ; establishing frame chain
    mov     fp, sp

    ; initializing GS cookie
    bl      __security_push_cookie

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

for a combined stack layout of



There is no change to the epilogue because variadic inbound register parameters don't need to be preserved across the call.

Another type of function that has a different prologue and epilogue is the lightweight leaf function. This is a function that does not catch any exceptions, does not use any stack (beyond the stack space provided by inbound stack-based parameters), and does not modify

any nonvolatile registers. These functions do not need to set up a stack frame or declare exception unwind codes. Conversely, the operating system assumes that any function that lacks exception unwind codes is a lightweight leaf function.[1] If the operating system needs to unwind an exception that occurs in a lightweight leaf function, it just unwinds to the address held in the *lr* register without updating any registers.

A third category of function is the shrink-wrapped function. This function starts out with a minimal stack frame (possibly none, posing as a lightweight leaf function), but conditionally optionally expands to a full stack frame. This is typically used for functions that have a fast early-exit.

And of course, for any of these functions, the `RET` could turn into a `B` or `BR` if a tail call optimization is in effect. Since the calling convention is caller-clean with many register-based parameters, tail call optimization doesn't require the tail-called function to have an identical signature as the calling function. The tail call is in play as long as the tail-called function's stack parameter size is less than or equal to the stack parameter size of the calling function.

Okay, so that's the prologue and epilogue. Next time, we'll look at some code patterns you'll see in the function body itself.

[1] This clause always catches out people who are trying to write Windows programs in assembly language. If you fail to declare unwind codes, then the operating system assumes you are a lightweight leaf function: It assumes that the return address is in the *lr* register, and it will continue stack walking at whatever address happens to be in that register, with whatever stack happens to be active, and with whatever values happen to be in the nonvolatile registers. This prevents the exception handlers installed by outer frames from running, or at least from running correctly. If you're lucky, the program is declared "corrupted, possible malware" by the kernel when it notices that the frame chain is nonsense. If you're not lucky, the handler is called with invalid registers and an invalid stack, and things spiral out of control. And if you're super-unlucky, an attacker may be able to use this for a remote code execution attack.

Raymond Chen

Follow