Let’s get modern: 64 bits

Quick tour of “x86-64”

Reversing 64 bits code is easy: tracing the code is much easier than tracing 32 bits code.

Let’s talk registers

- Registers have been extended from 32 to 64 bits
  - EAX -> RAX, ESP -> RSP, etc…
- Some new GPRs are now available
  - R8 to R15
    - 64 bits registers
    - These register can be accessed at byte, word and double word level by appending b, w, d to the register name

Calling convention (functions)

- Function calling convention also changed
- First 6 parameters are now passed in registers
  - Register order goes as: RDI, RSI, RDX, RCX, R8, R9
  - If more that 6 parameters are required, parameters (7th onward) are pushed on the stack in reverse order
- The following registers are “scratch” registers, other registers need to be preserved
  - RAX, RDI, RSI, RDX, RCX, R8, R9, R10, R11
- When calling, RAX is used to indicate the number of vector registers used (can safely ignore for now and simply set RAX to 0 before calling)

16 bytes alignement

- You’re also supposed to keep a 16 bytes alignement for the stack
- This means your stack pointer should always end in “0” when looking at it in hexadecimal
- Following the stack frame creation technique should ensure you’re 16 bytes aligned:
  - push rbp
  - mov rbp, rsp
- Subtractions to stack pointers should then be on 16 bytes boundaries
  - if you only need 4 bytes, you still sub 16 bytes from the stack pointer
    - Do “sub rsp, 0x10” and NOT “sub rsp, 0x04”
  - This will ensure you’re always 16 bytes aligned

But why?

Performance!

- Simply said (omitting a lot of details):
- Cache system will work better with 16 bytes aligned data since a “block” of data will fit the size of a cache line. This will prevent “crossing” cache line boundary in accessing data and will make the performance of memory access much better.
- 16 bytes is also the size of XMM (SIMD) registers.

- On 32 bits systems, this alignment is 4 bytes which makes it almost “forgettable”

Hello 64 bits functions

```
section .data
        hello:
                db "Hello world %d",0xa, 0

section .text
extern printf

global main

main:

        push rbp
        mov rbp, rsp

        mov rdi, hello
        mov rsi, 0xFF
        xor rax, rax
        call printf


        xor rax, rax
        pop rbp
        ret
```

These rules are true for systems using System V ABI (Linux, MacOS, …).

Windows uses a different calling convention (RCX, RDX, R8, R9 then stack).
https://msdn.microsoft.com/fr-ca/library/ms235286.aspx

Calling convention (system calls)

- System calls calling convention changed
- A **maximum of 6 parameters** is to be used
  - Register order is RDI, RSI, RDX, R10, R8, R9
  - If you forget simply do “man syscall”
- System **call number** is passed into **RAX**
- After syscall, **RAX** contains the **result** of the syscall
- System **calls numbers changed**, full list for 64 bits is located:
  - /usr/include/x86_64-linux-gnu/asm/unistd_64.h
- Syscall now uses the instruction “**syscall**”
  - No more int 0x80

Hello 64 bits system calls

```
section .data
        hello:
                db "Hello world",0xa, 0

section .text
extern printf

global main

main:

        push rbp
        mov rbp, rsp

        mov rdi, 0x01  ; stdout
        mov rsi, hello ; buffer
        mov rdx, 0x0c  ; buffer size
        mov rax, 0x01  ; syscall number 1 (write)
        syscall

        xor rax, rax
        pop rbp
        ret
```

System V 64 bits ABI can be downloaded here

https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

Obviously other differences exists. For example, the structures used for paging are somewhat different. Most of these differences are of little interest here.

Let’s code!