Applied Reverse Engineering: Basic Architecture (Summarized)

Credits to: Daax in https://revers.engineering/applied-re-basic-architecture/

Summarized by: Youmna Elmezayen

Introduction: This article will involve dissecting a C program in execution which calls a Windows API to get the computer name, implements a custom strlen function, and prints out the computer name and resulting name length. This process will involve compiling it and diving into the resulting assembly.

Compilation process: This process involves running the C compiler on the written program. The C compiler is responsible for 4 main stages: preprocessing, compiling, assembling, and linking. The first pass of the compiler does the preprocessing which is to expand included files into the source code, preprocesses directives, and does other conditional compilation work. The second pass is the compiling. This can be done either by translating the source language C into an intermediate language (assembly), or by using what's called an integrated assembler which translates the assembly into machine language from within the compiler and outputting the machine code directly. The next stage is the assembling, which is taking the output of the compiler (in case it is still in assembly) and translating it into object code, or machine code. Lastly, the linking. The linking is the process of taking different modules of the output object code and rearranging it together to get the executable. This stage also links the libraries used in the program.

The written C:

```c
int main(int argc, char **argv, char **envp)
{
    CHAR ComputerName[MAX_COMPUTERNAME_LENGTH + 1];
    DWORD MaxCopied;

    memset(ComputerName, 0, MAX_COMPUTERNAME_LENGTH + 1);

    MaxCopied = sizeof(ComputerName);
    GetComputerNameExA(ComputerNamePhysicalDnsHostname, ComputerName, &MaxCopied);

    printf("Computer Name: %s\n", ComputerName);
    printf("Computer Name Length: %d\n", MaxCopied);

    return 0;
}
```

The assembly post-compilation:

```
main    PROC
; File applied_reverse_engineering.c
$LN3:
        mov     QWORD PTR [rsp+24], r8
        mov     QWORD PTR [rsp+16], rdx
        mov     DWORD PTR [rsp+8], ecx
        push    rdi
        sub     rsp, 64                                 ; 00000040H
        mov     rax, QWORD PTR __security_cookie
        xor     rax, rsp
        mov     QWORD PTR __$ArrayPad$[rsp], rax
; Line 11
        lea     rax, QWORD PTR ComputerName$[rsp]
        mov     rdi, rax
        xor     eax, eax
        mov     ecx, 16
        rep stosb
; Line 13
        mov     DWORD PTR MaxCopied$[rsp], 16
; Line 14
        lea     r8, QWORD PTR MaxCopied$[rsp]
        lea     rdx, QWORD PTR ComputerName$[rsp]
        mov     ecx, 5
        call    QWORD PTR __imp_GetComputerNameExA
; Line 16
        lea     rdx, QWORD PTR ComputerName$[rsp]
        lea     rcx, OFFSET FLAT:$SG96224
        call    printf
; Line 17
        mov     edx, DWORD PTR MaxCopied$[rsp]
        lea     rcx, OFFSET FLAT:$SG96225
        call    printf
; Line 19
        xor     eax, eax
; Line 20
        mov     rcx, QWORD PTR __$ArrayPad$[rsp]
        xor     rcx, rsp
        call    __security_check_cookie
        add     rsp, 64                                 ; 00000040H
        pop     rdi
        ret     0
main    ENDP
```

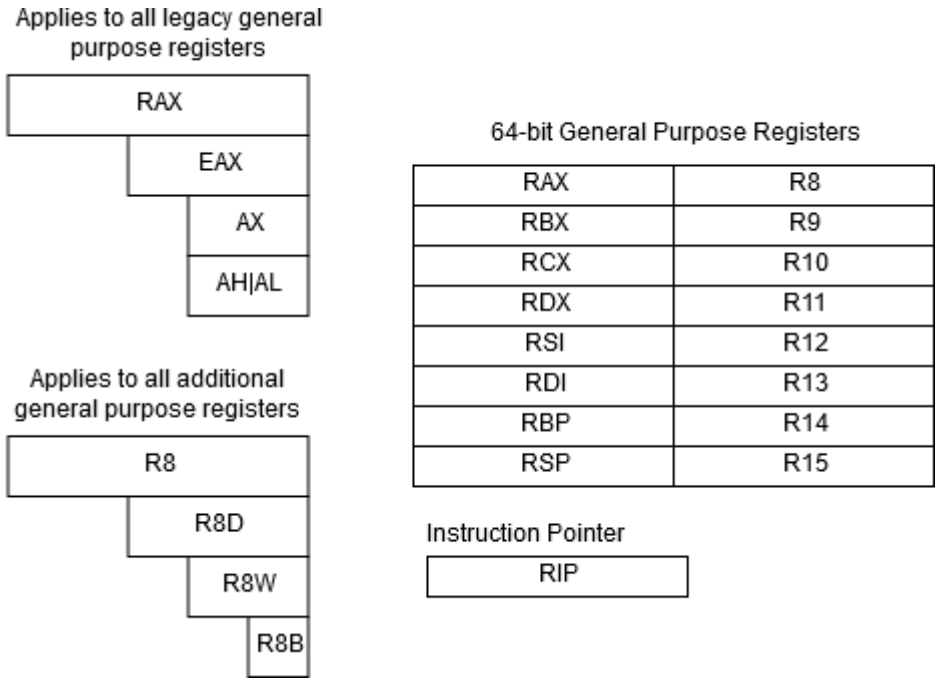
Note: This isn't the full compiled code; this is just the main function that was written by the programmer.

This is compiled on the x86 architecture. The x86 architecture has hundreds of instructions that have tens of variations. Each instruction is made up of 2 main things: a mnemonic and an operand (or 2, sometimes 3). A mnemonic is the actual function or operation that will be executed, and it will execute on its given operands, which are generally given as registers (more on registers soon). An example of such instruction is the "mov rdi, rax", which basically says *move* whatever is stored in rax (the source) into rdi (the destination).

Microarchitecture: Any processor is made up of the same core components, namely an ALU (arithmetic logic unit), some sort of memory that is faster and closer to the processor than the main memory (cache), a branch predictor for conditional instructions, and finally a register file, which is the set of existing registers.

The register file is a set of registers that are used to store information needed by the processor. Instead of the processor having to interface with the memory every time it needs a piece of information, it can move the data it needs frequently closer to it and in turn optimize

running time. On the Intel 64 architecture, the register file contains 16 general purpose registers, each register's capacity being 64-bits. General purpose registers are used to perform basic operations such as data movement or control flow operations. The sizes of registers are usually referred to in terms of words: word, doubleword, etc. A word is defined as 16 bits of data, so that makes a double word 32 bits and so on. Their sizes are also denoted in terms of bytes. So, a word has 2 bytes.



As mentioned, the 64-bit architecture has 64-bit registers. An older architecture is the 32-bit architecture which has 32-bit registers. To maintain compatibility with 32-bit architectures, some registers can be accessed with varying sizes. For example, the RAX register can be accessed as 64 bits, as 32 bits (EAX), as 16 bits (AX), or as 8 bits (either AH or AL, which stand for high byte or low byte, respectively). This is useful because there are many instances where the value stored would be too small for the whole 64-bit space, so we can then access it directly with the other variations. Moreover, we can store data only in the lower words or bytes without affecting the rest of the register. The same applies to the rest of the general-purpose registers mentioned above.

A special register called RIP is used to point to the next instruction to be executed. It is automatically incremented after each instruction. It can sometimes move backwards if there are conditional instructions that specify this.

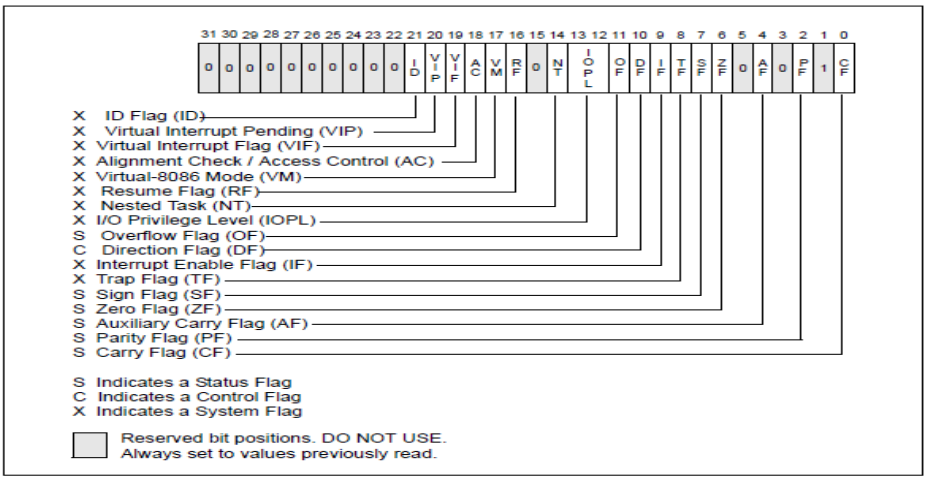


Figure 3-8. EFLAGS Register

EFLAGS register is a special kind of register that holds what are called flags that the system can read and update. These flags describe the status in the running program. Some are updated in arithmetic operations, while others are used to control OS related operations.

The S (status) flags:

- Zero flag: set (=1) if the result of the current operation is 0.

  Example in assembly:

```
    mov rax, 510
    mov rbx, 511
    sub rax, rbx
    jnz zf_not_set  -> jump to label zf not set if result of sub is not 0
    lea rcx, offset zf_set_string
    call printf
    jmp end
.zf_not_set:
    lea rcx, offset zf_not_set_string
    call printf
.end:
    Ret
```

- Sign flag: set (=1) if the current operation is on signed data and the result is negative.
  Example in assembly:

```
    mov rax, 1
    mov rbx, 1000
    sub rax, rbx
    jge sf_not_set -> jump to label sf not set if result of sub is greater
than or equal 0 (not negative)
    lea rcx, sf_set_string
    call printf
    jmp end
.sf_not_set:
    lea rcx, sf_not_set_string
    call printf
.end:
    xor eax, eax
    ret
```

Note: There are jump instructions that are based on the status flags. They are called ‘jump if condition is met’ or Jcc for short.

- Carry flag: set (=1) if the current operation is arithmetic and it generated a carry or borrow. It is also set when there is overflow (goes over the maximum value that can be stored in the given location).