리버싱 정리

x86-64 아키텍처 - 레지스터

- 레지스터는 CPU 내부의 저장장치로, CPU가 빠르게 접근하여 사용할 수 있다.
    - 산술 연산에 필요한 데이터를 저장하거나, 주소를 저장하고 참조하는 등 다양한 용도로 사용된다.
        - x64 아키텍처에는 범용 레지스터, 세그먼트 레지스터, 명령어 포인터 레지스터, 플래그 레지스터가 존재한다.

범용 레지스터

- 범용 레지스터는 주용도는 있으나, 그 외 임의의 용도로도 사용될 수 있는 레지스터이다.
    - x86-64에서 각각의 범용 레지스터는 8바이트를 저장할 수 있으며, 부호 없는 정수를 기준으로 2^64 -1 까지 나타낼 수 있다.

- 자주 쓰이는 범용 레지스터들의 주용도는 아래와 같다.
    - 여기 서술된 레지스터 외에도 x64에는 r8, r9, ..., r15까지의 범용 레지스터가 더 존재한다.

이름	주용도
rax (accumulator register)	함수의 반환 값
rbx (base register)	x64에서는 주된 용도 없음
rcx (counter register)	반복문의 반복 횟수, 각종 연산의 시행 횟수
rdx (data register)	x64에서는 주된 용도 없음
rsi (source index)	데이터를 옮길 때 원본을 가리키는 포인터
rdi (destination index)	데이터를 옮길 때 목적지를 가리키는 포인터
rsp (stack pointer)	사용중인 스택의 위치를 가리키는 포인터
rbp (stack base pointer)	스택의 바닥을 가리키는 포인터

- rax (accumulator register) : 함수의 반환 값

- rbx (base register) : x64에서는 주된 용도 없음
- rcx (counter register) : 반복문의 반복 횟수, 각종 연산의 시행 횟수
- rda (data register) : x64에서는 주된 용도 없음
- rsi (source index) : 데이터를 옮길 때 원본을 가리키는 데이터
- rdi (destination index) : 데이터를 옮길 때 목적지를 가리키는 데이터
- rsp (stack pointer) : 사용 중인 스택의 위치를 가리키는 포인터
- rbp (stack base pointer) : 스택의 바닥을 가리키는 포인터

세그먼트 레지스터

- x64 아키텍처에는 cs, ss, ds, es, fs, gs 총 6가지 레지스터가 존재하며, 각 레지스터의 크기는 16바이트이다.
    - 세그먼트 레지스터는 x64로 아키텍처가 확장되면서 용도에 큰 변화가 생긴 레지스터이다.
        - 과거 IA-32, IA-16에서는 세그먼트 레지스터를 이용하여 사용 가능한 물리 메모리의 크기를 키우려고 했다.
            - 예를 들어, IA-16에서는 어떤 주소를 cs:offset이라고 한다면, 실제로는 cs<<4 + offset의 주소를 사용하여 16비트 범위에서 접근할 수 없는 주소에 접근할 수 있었다.
                - 당시에는 범용 레지스터의 크기가 작아서 사용 가능한 메모리의 주소 폭이 좁았지만,
                    - x64에서는 사용 가능한 주소 영역이 굉장히 넓기 때문에 이런 용도로는 거의 사용되지 않는다.

- 현대의 x64에서 cs, ds, ss 레지스터는 코드 영역과 데이터, 스택 메모리 영역을 가리킬 때 사용되고, 나머지 레지스터는 운영체제 별로 용도를 결정할 수 있도록 범용적인 용도로 제작된 세그먼트 레지스터이다.

명령어 포인터 레지스터

- 프로그램의 코드는 기계어로 작성되어 있다.

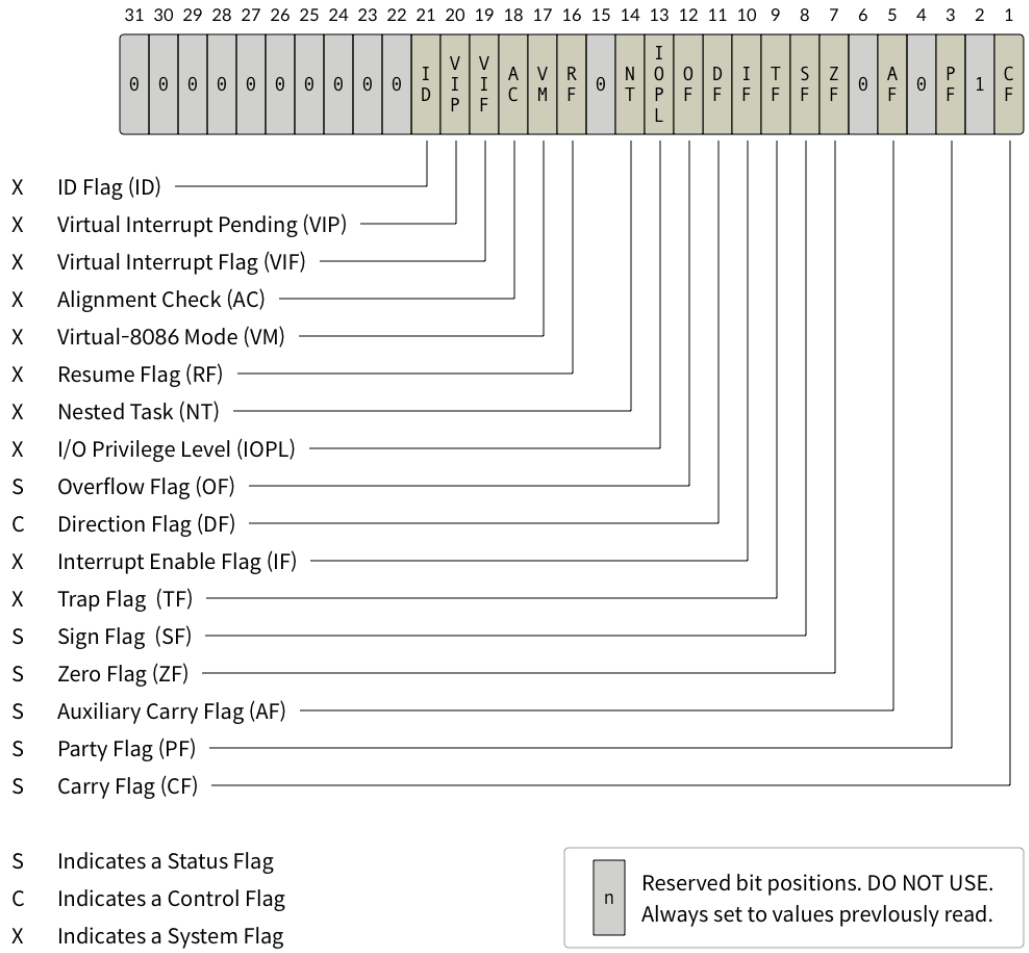
- 이 중에서 CPU가 어느 부분의 코드를 실행할지 가리키는게 명령어 포인터 레지스터의 역할이다.
    - x64 아키텍처의 명령어 레지스터는 rip이며, 크기는 8바이트이다.

플래그 레지스터

- 플래그 레지스터는 프로세서의 현재 상태를 저장하고 있는 레지스터이다.
    - x64에서는 RFLAGS라고 불리는 64비트 크기의 플래그 레지스터가 존재하며, 과거 16비트 플래그 레지스터가 확장된 것이다.
        - 깃발을 올리고 내리는 행위로 신호를 전달하듯, 플래그 레지스터는 자신을 구성하는 여러비트들로 CPU의 현재 상태를 표현한다.
    - 플래그 레지스터의 여러 플래그 비트들 중 앞으로 주로 접하게 될 것들은 아래와 같다.

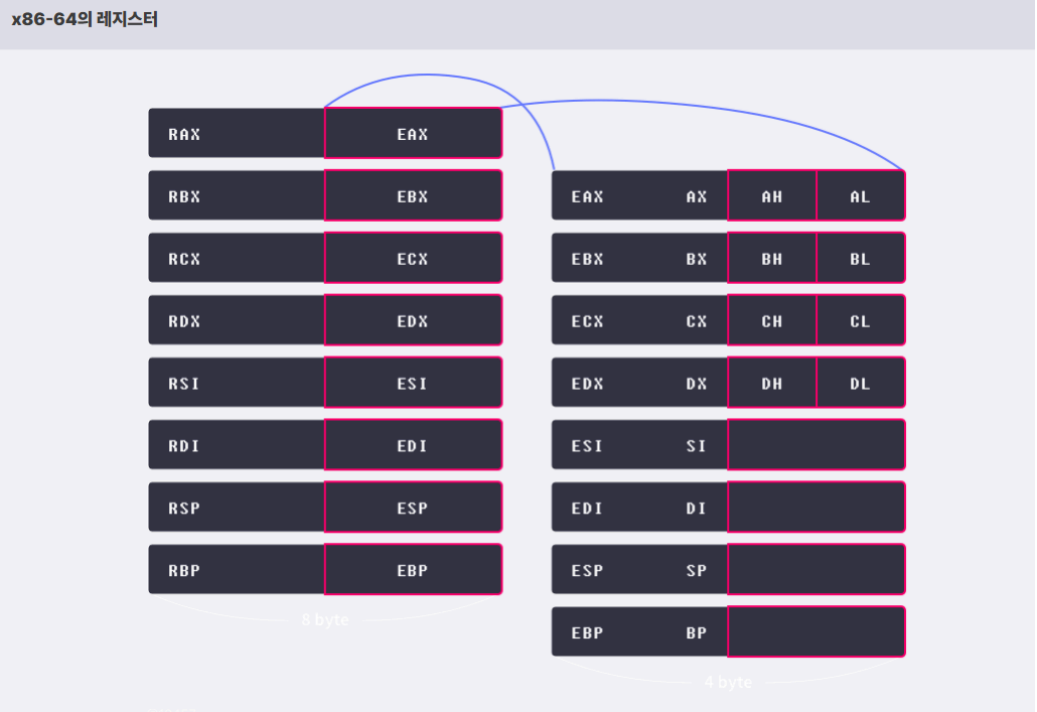
플래그	의미
CF (Carry Flag)	부호 없는 수의 연산 결과가 비트의 범위를 넘을 경우 설정 됩니다.
ZF (Zero Flag)	연산의 결과가 0일 경우 설정 됩니다.
SF (Sign Flag)	연산의 결과가 음수일 경우 설정 됩니다.
OF (Overflow Flag)	부호 있는 수의 연산 결과가 비트 범위를 넘을 경우 설정 됩니다.

        - RFLAGS는 64비트이므로, 최대 64개의 플래그를 사용할 수 있지만, 오른쪽의 20여개의 비트만 사용한다.



레지스터 호환

- 앞에서 x86-64 아키텍처는 IA-32의 64비트 확장 아키텍처이며, 호환이 가능하다고 했다.
  - IA-32에서 CPU의 레지스터들은 32비트 크기를 가지며, 이들의 명칭은 각각, eax, ebx, ecx, edx, esi, edi, esp, ebp였다.
    - 호환성을 위해 해당 레지스터들은 x86-64에서도 그대로 사용이 가능하다.
  - 앞서 소개한, rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp가 이들의 확장된 형태이며, eax, ebx 등은 확장된 레지스터의 하위 32비트를 나타낸다.
    - 예를 들어, eax는 rax의 하위 32비트를 나타낸다.
  - 또한, 마찬가지로, 과거 16비트 아키텍처인 IA-16과의 호환을 위해 ax, bx, cx, dx, si, di, sp, bp는 eax, ebx, ecx, edx, esi, edi, esp, ebp의 하위 16비트를 가리킨다.
    - 이들 중 몇몇은 다시 상위 8비트, 하위 8비트로 나뉜다.



Windows Memory Layout

섹션

- 윈도우의 PE 파일은 PE 헤더 1개와 1개 이상의 섹션으로 구성되어 있다.
  - 여기서 섹션이란, 유사한 용도로 사용되는 데이터가 모여있는 영역이다.
    - 예를 들어, ".text"섹션에는 PE의 코드가 적혀있고, ".data"에는 PE가 실행 중에 참조하는 데이터가 적혀있다.
  - 섹션에 대한 정보는 PE 헤더에 적혀 있다.
    - PE 헤더가 저장되는 섹션과 관련된 데이터 중, 중요한 것은 아래와 같다.
      - 섹션의 이름
      - 섹션의 크기
      - 섹션이 로드될 주소의 오프셋
      - 섹션의 속성과 권한
    - 윈도우는 PE를 실행할 때, 해당 정보를 참조하여 PE의 각 섹션들을 가상 메모리의 적절한 세그먼트에 매핑한다.

- PE에 필수로 존재해야하는 섹션이 정해진 것은 아니지만, “.text”, “.data”, “.rdata”섹션이 일반적으로 사용된다.

.text

- .text 섹션은 실행 가능한 기계 코드가 위치하는 영역이다.
  - 프로그램이 동작하려면 코드를 실행할 수 있어야 하므로, 해당 세그먼트에는 읽기 권한과 실행 권한이 부여된다.
    - 반면, 쓰기 권한이 있으면 공격자가 악의적인 코드를 삽입하기가 쉬워지므로, 대부분의 현대 운영체제는 해당 세그먼트에 쓰기 권한을 제거한다.

```
int main() {return 31337;}
```

- 위에서 정수 31337을 반환하는 main( ) 함수가 컴파일되면

```
554889e5b8697a00005dc3
```

  - 라는 기계 코드로 컴파일 되는데, 해당 기계 코드가 코드 세그먼트에 위치하게 된다.

.data

- .data 섹션에는 컴파일 시점에 값이 정해진 전역 변수들이 위치한다.
  - CPU가 해당 섹션의 데이터를 읽고 쓸 수 있어야 하므로, 읽기 / 쓰기 권한이 부여된다.
- 아래는 .data 섹션에 포함되는 여러 데이터의 유형이다.

```
int data_num = 31337;
char data_rwstr[] = "writable_data";       // data

int main() { ... }
```

.rdata

- .rdata 섹션에는 컴파일 시점에 값이 정해진 전역 상수와 참조할 DLL 및 외부 함수들의 정보가 저장된다.
    - CPU가 해당 섹션의 데이터를 읽을 수 있어야 하므로 읽기 권한은 부여되지만, 쓰기는 불가능하다.

```
const char data_rostr[] = "readonly_data";
char *str_ptr = "readonly";  // str_ptr은 .data, 문자열은 .rdata

int main() { ... }
```

- 위에는 .rdata 섹션에 포함되는 여러 데이터의 유형이다.
    - 주의 깊게 살펴봐야할 변수는 str_ptr이다.
        - str_ptr은 전역 변수로서 .data에 위치하지만, “readonly”는 상수 문자열로 취급되어 .rdata에 위치한다.
            - 과거에는 참조할 DLL과 외부 함수들을 .idata 섹션에 저장하였으나, 최근에는 대부분 .rdata 섹션에 저장한다.

섹션이 아닌 메모리

- 윈도우의 가상 메모리 공간에는 섹션만 로드되는 것이 아니다.
    - 프로그램 실행에 있어 필요한 스택과 힙 역시 가상 메모리 공간에 적재된다.

스택

- 윈도우즈 프로세스의 각 쓰레드는 자신만의 스택 공간을 가지고 있다.
    - 보통 지역 변수나 함수의 리턴 주소가 저장된다.
        - 해당 영역은 자유롭게 읽고 쓸 수 있어야 하므로, 읽기 / 쓰기 권한이 부여된다.
            - 참고로 스택에 대해서 “아래로 자란다.”라는 표현을 종종 사용하는데, 이는 스택이 확장될 때, 기존 주소보다 낮은 주소로 확장되기 때문이다.
                - 아래 코드에서는 지역변수 choice가 스택에 저장되게 된다.

```
void func() {
  int choice = 0;
```

```
  scanf("%d", &choice);

  if (choice)
    call_true();
  else
    call_false();

  return 0;
}
```

힙

- 힙은 프로그램이 여러 용도로 사용하기 위해 할당 받는 공간이다.
  - 따라서 모든 종류의 데이터가 저장될 수 있다.
    - 스택과 다른 점은 비교적 스택보다 큰 데이터도 저장할 수 있고, 전역적으로 접근이 가능하도록 설계되었단 점이다.
      - 또한 실행 중 동적으로 할당받는 점 역시 다르다.
        - 권한은 보통은 데이터를 읽고 쓰기만 하기 때문에 읽기 / 쓰기 권한만을 가지나, 상황에 따라 실행 권한을 가지는 경우도 존재한다.

```
int main() {
  int *heap_data_ptr =
      malloc(sizeof(*heap_data_ptr));  // 동적 할당한 힙 영역의 
  *heap_data_ptr = 31337;              // 힙 영역에 값을 씀
  printf("%d\n", *heap_data_ptr);  // 힙 영역의 값을 사용함
  return 0;
}
```

        - 위의 예제 코드는 heap_data_ptr에 malloc()으로 동적할당한 영역의 주소를 대입하고, 해당 영역에 값을 쓴다.
          - heap_data_ptr은 지역변수이므로, 스택에 위치하며, malloc으로 할당받은 힙 세그먼트의 주소를 갖는다.