201716905 김강민

Stack overflow

IT 정보공학과 BCG LAP

· 메모리 구조

- 커널 공간은 사용자가 접근 불가

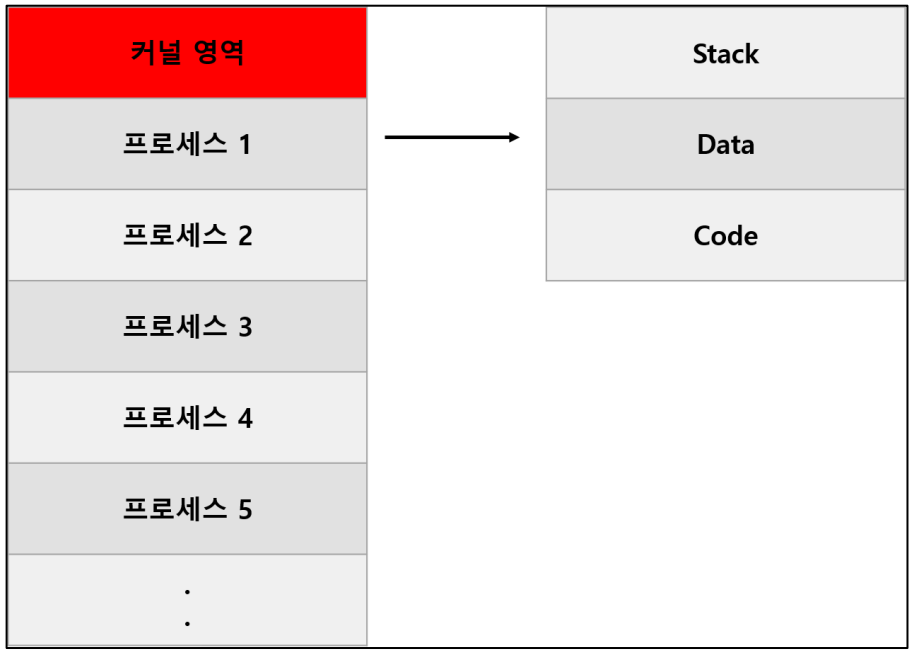
- Stack : 지역변수, 매개변수 등 프로그램이 사용하는 임시 메모리 영역

- Heap : 동적 메모리가 저장되는 영역

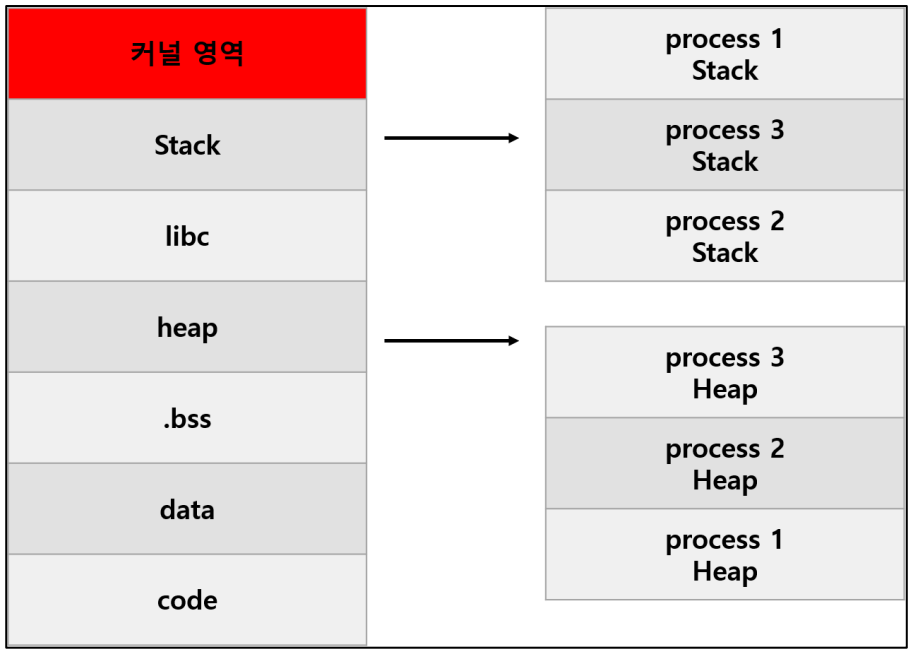
- Data : 초기화 여부에 따라 나눠 저장되며 데이터를 저장하는 영역

- Code: 코드가 여기에 저장되며, 전역 상수 저장

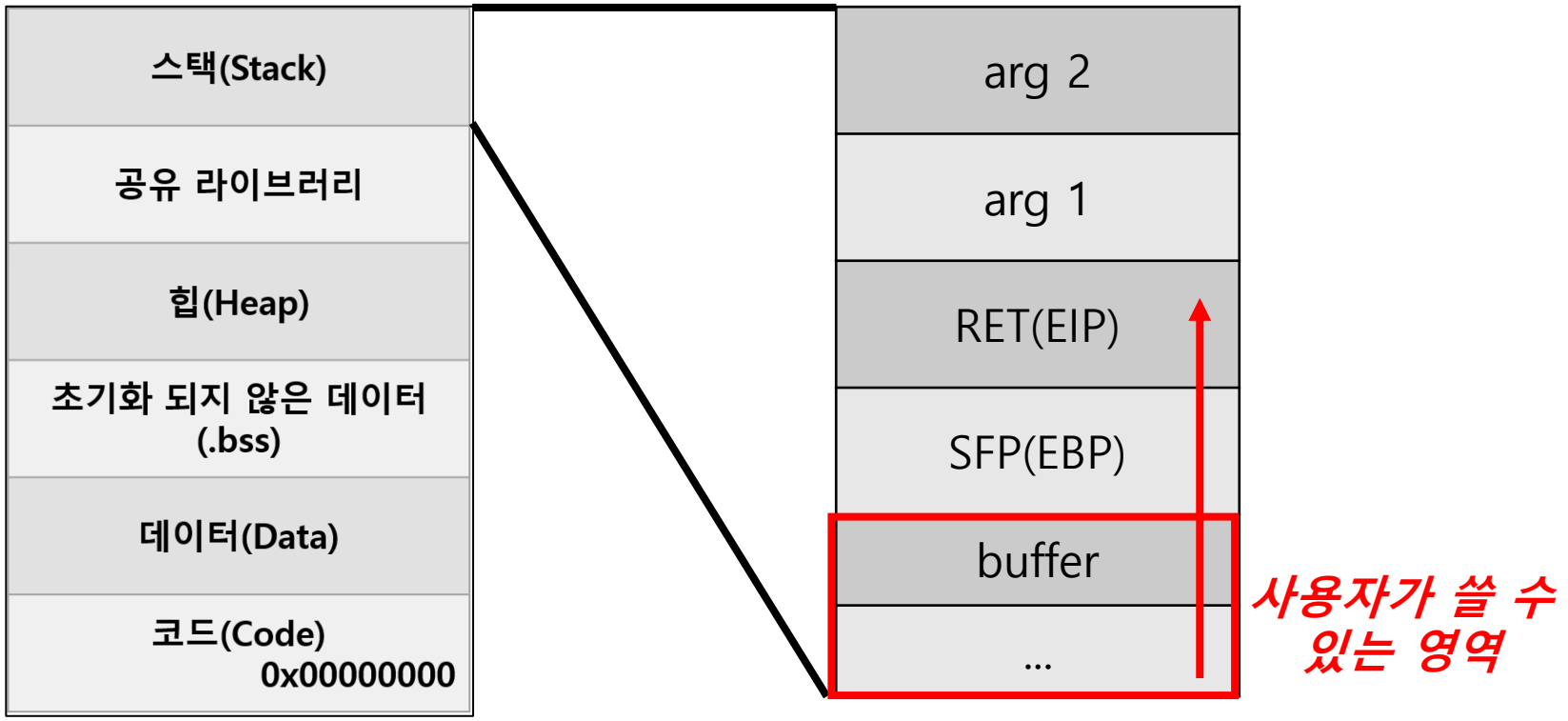
· 연속 메모리 구조(예전)



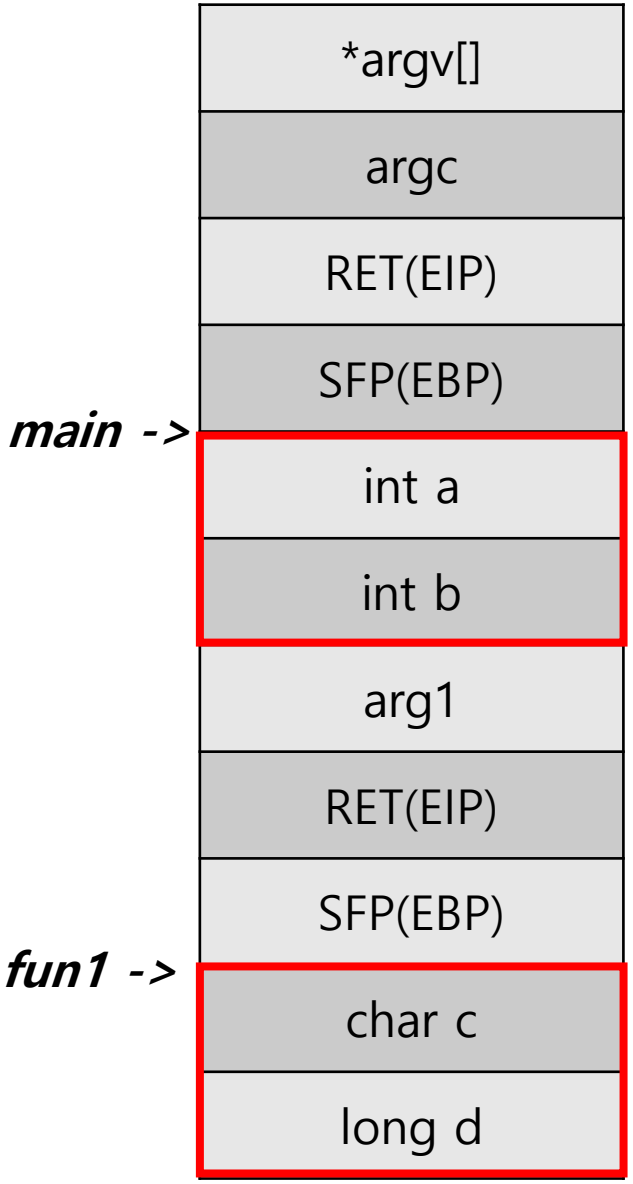
· 비연속 메모리 구조(현재)



· Stack overflow



01 스택 오버 플로우

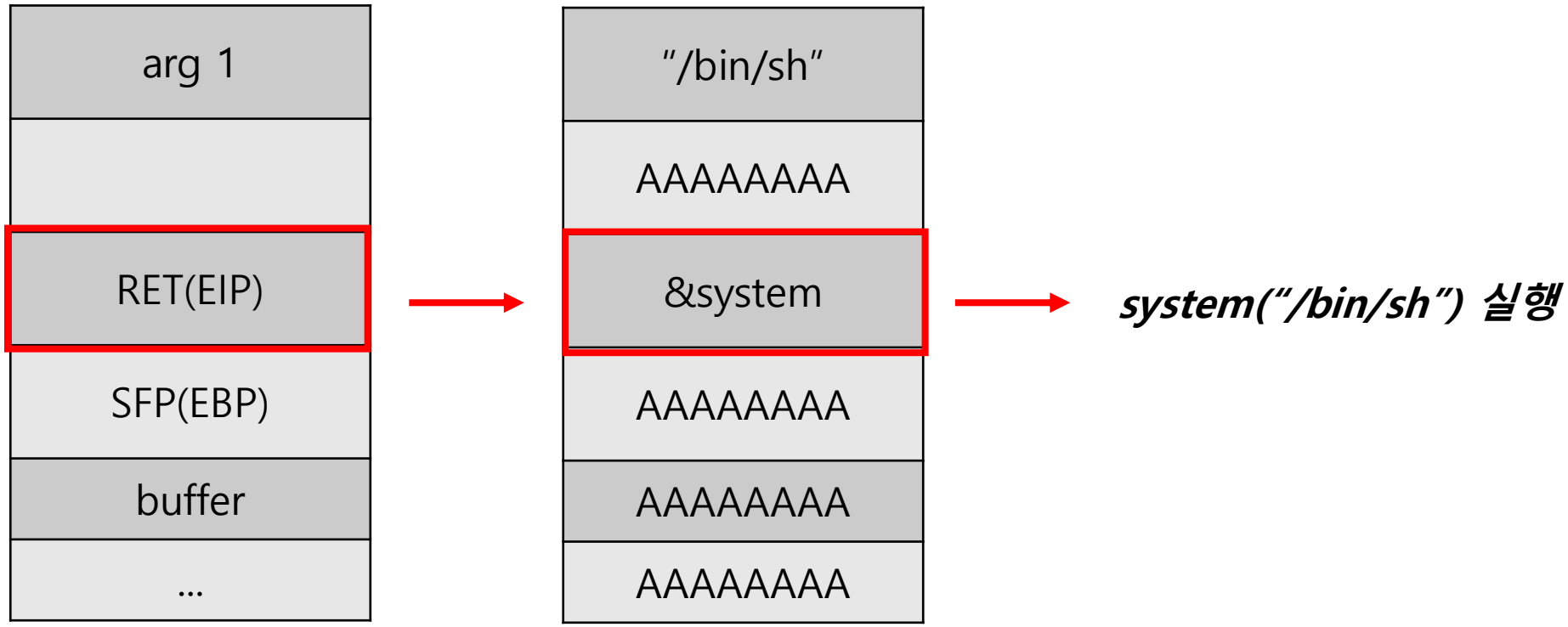


- **SFP와 RET**

- 함수를 호출할 때마다 호출한 함수의 stack 공간을 할당

- SFP : Stack Frame Pointer, 각 함수의 스택 시작점을 저장

- RET : 함수가 종료되고 실행할 명령어의 주소를 저장


arg 1
RET(EIP)
SFP(EBP)
buffer
...
"/bin/sh"
AAAAAAAA
&system
AAAAAAAA
AAAAAAAA
AAAAAAAA
system("/bin/sh") 실행

- **NOP Sled**

- 스택 영역에 작성 쉘 코드를 실행시키는 고전적 방법

- EIP가 ₩x90을 읽으면 무시하고 다음 4 Byte를 읽는 성질 이용

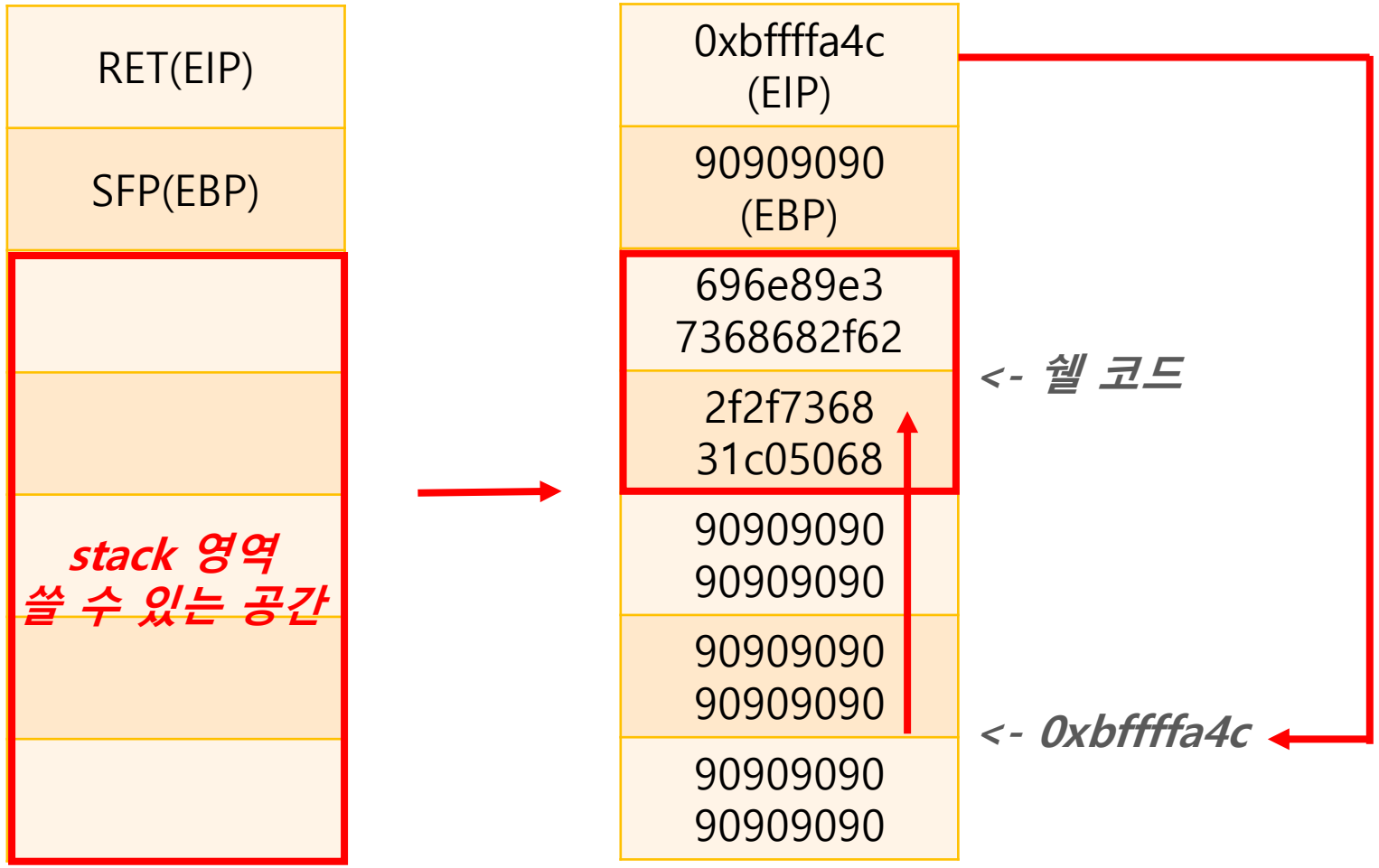
- payload에 ₩x90을 넣은 후 쉘 코드를 작성하면, ₩x90을 타고 쉘 코드까지 가서 실행

· NOP sled 필요조건

보호 기법	상태
RELRO	no RELRO
Stack guard	No canary found
NX bit	NX disable
PIE	No PIE

1. 보호기법 해제
2. 쉘 코드를 실행 시킬 수 있는 충분한 공간
3. 스택 오버 플로우나 임의 주소 쓰기 취약점 존재

· NOP sled 동작과정



- **환경 변수를 이용한 쉘 코드 실행**

- 스택 영역에 작성 쉘 코드를 실행시키는 고전적 방법

- stack overflow나 임의 주소 쓰기 취약점을 이용

- EIP에 미리 입력해둔 shell code가 담긴 환경 변수를 이용

· 환경변수 공격 필요조건

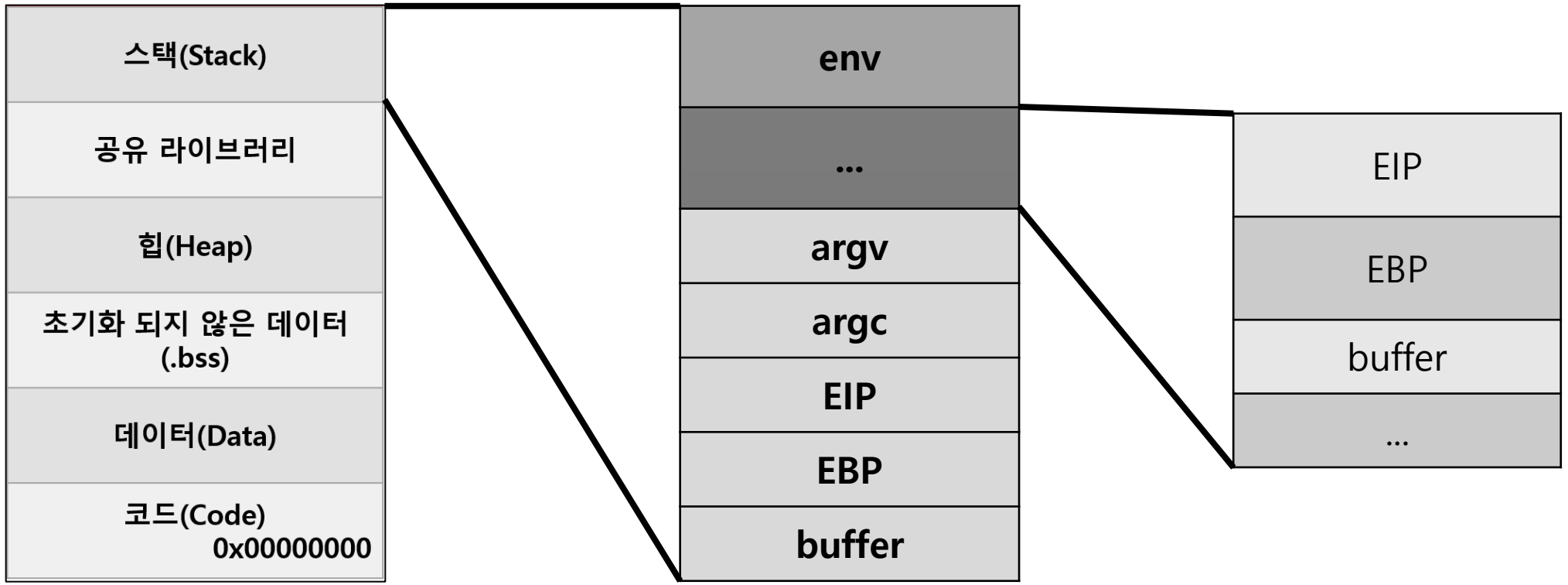
보호 기법	상태
RELRO	no RELRO
Stack guard	No canary found
NX bit	NX disable
PIE	No PIE

1. 보호기법 해제

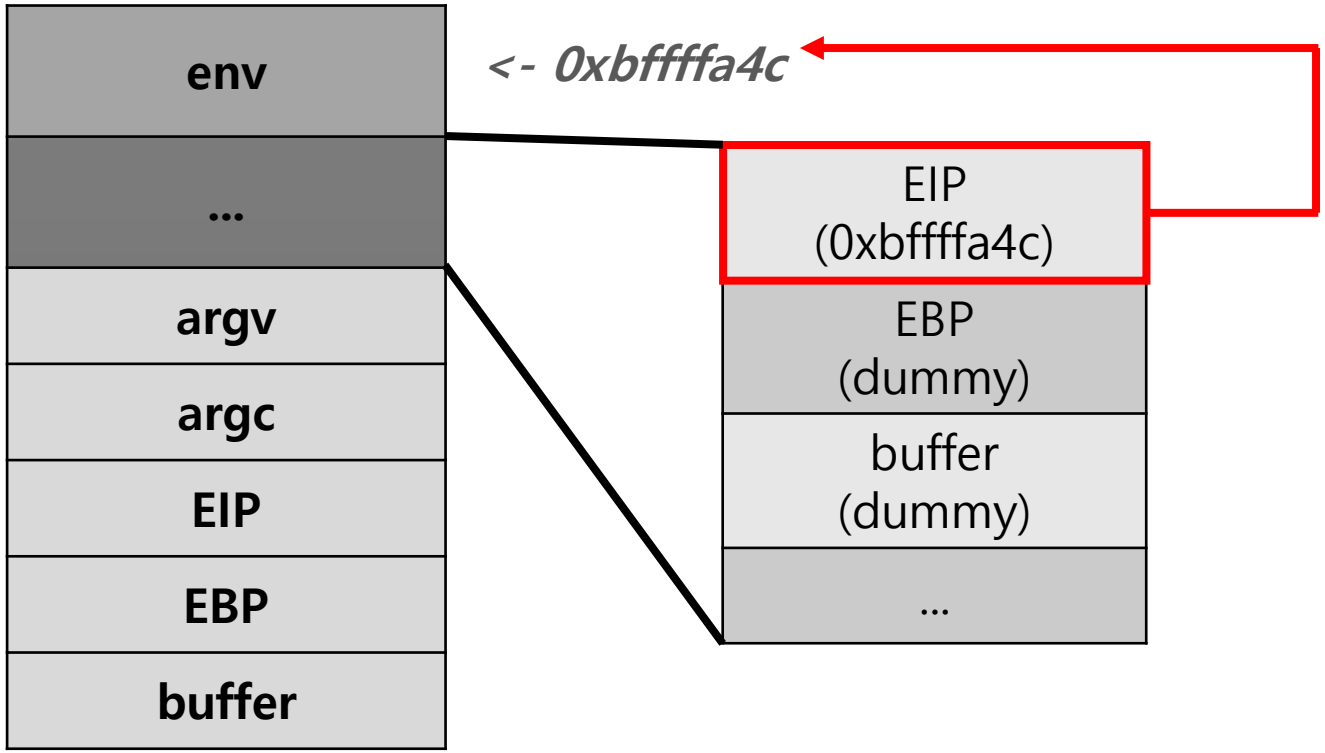
2. 스택 오버 플로우나 임의 주소 쓰기 취약점 존재

-> NOP sled처럼 stack 영역의 큰 공간을 요구하지 않는다.

· 환경 변수


스택(Stack)
공유 라이브러리
힙(Heap)
초기화 되지 않은 데이터 (.bss)
데이터(Data)
코드(Code)
0x00000000
env
...
argv
argc
EIP
EBP
buffer
EIP
EBP
buffer
...

· 환경 변수 공격 동작 과정


env
...
argv
argc
EIP
EBP
buffer
<- 0xbffffa4c
EIP
(0xbffffa4c)
EBP
(dummy)
buffer
(dummy)
...

- **Nx bit**

- 스택 영역에서 명령어 실행 불가

- NOP sled, 환경 변수처럼 스택 영역에 쉘 코드를 작성하여 실행하는 취약점 방어

- **Return to libc (RTL)**

- return address에 원하는 함수를 써서 EIP가 실행시키게 하는 방법

- Nx bit를 대응하기 위해 생김

- RTL Chaining과 ROP의 기반이 되는 기술

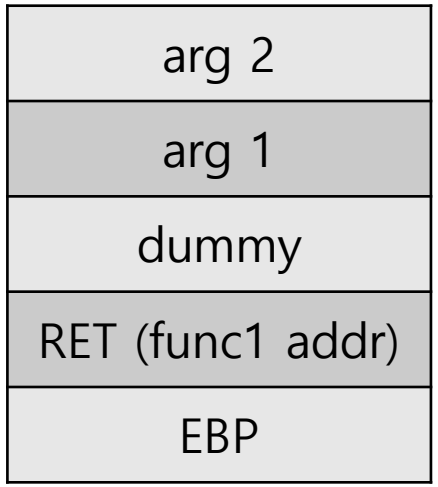
· RTL 필요조건

보호 기법	상태
RELRO	partial RELRO
Stack guard	No canary found
NX bit	NX enable
PIE	No PIE

1. Nx bit를 제외한 보호기법 해제

2. 스택 오버 플로우가 일어나는 환경

3. libc의 주소 획득

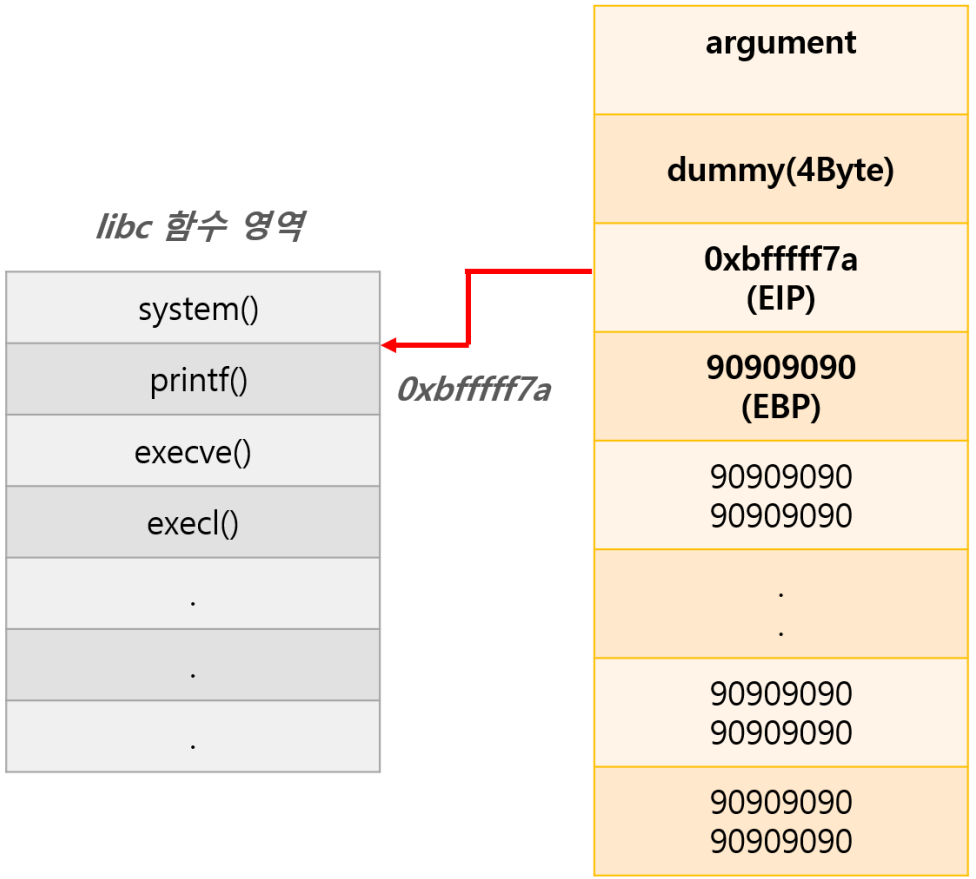


*** func1(arg1, arg2)**

· Return address & argument

- EBP를 기준으로 +4 주소는 return address의 주소
- 인자는 EBP로부터 12(0xC)만큼 떨어진 주소에 값을 입력
- 4만큼 추가해서 인자 값 추가

· RTL 동작과정


libc 함수 영역
system()
printf()
execve()
execl()
.
.
.
0xbfffff7a
argument
dummy(4Byte)
0xbfffff7a
(EIP)
90909090
(EBP)
90909090
90909090
.
.
90909090
90909090
90909090
90909090

- **RTL Chaining**

- RTL 기법을 연속적으로 수행하여 원하는 함수를 연속적으로 호출하는 기법

- PPR 가젯을 사용 (64bit binary에는 없을 수도 있음)

· RTL chaining 필요조건

보호 기법	상태
RELRO	prtial RELRO
Stack guard	No canary found
NX bit	NX enable
PIE	No PIE

1. Nx bit를 제외한 보호기법 해제

2. 스택 오버 플로우가 일어나는 환경

3. libc의 주소 획득

· func2() 호출 전 stack 상황 – func1의 stack

buffer	EBP	RET	dummy	func2 arg1	func2 arg2
		func2 addr		func2 arg1	func2 arg2

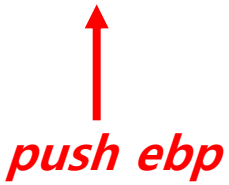
· func2() 호출 후 stack 상황 – func2의 stack

buffer	buffer	EBP	RET	dummy	func2 arg1
		func2 addr	dummy	func1 arg1	func1 arg2

· dummy가 다음 RET이 되는 이유

fun1	buffer	EBP	RET	dummy
fun2	buffer	buffer	EBP	RET

```
Dump of assembler code for function main:
   0x00000000000011c5 <+0>:     push   rbp
   0x00000000000011c6 <+1>:     mov    rbp,rsp
   0x00000000000011c9 <+4>:     sub    rsp,0x20
```



push ebp

- 호출이 끝난 함수는 RET까지의 주소를 사용
- 다음 push 명령어 수행 시 RET이었던 주소에 값이 써진다.
- 함수는 시작하면 push ebp를 수행
- ebp 다음 주소는 RET으로 사용

· stack overflow를 통해 func2 작성

buffer	EBP	RET	dummy	func2 arg1	func2 arg2
		func2 addr	func3 addr	fun2 arg1	fun2 arg2

· func2 함수 호출 종료 후 func 3 호출

buffer	buffer	EBP	RET	dummy	func2 arg1
			func3 addr	func2 arg1	func2 arg2

· func3 호출 후 stack 상황

buffer	buffer	buffer	EBP	RET	func2 arg1
				func2 arg1	func2 arg2

-> 이런 경우 func3가 실행 될 경우, 인자는 func2의 인자가 사용되고 RET 또한 func2의 인자가 되어 연속 호출 뿐만 아니라 원하는 인자로 호출이 불가능하다.

- **PPR 가젯**

```
1616:       41 5d                   pop    %r13
1618:       41 5e                   pop    %r14
161a:       41 5f                   pop    %r15
161c:       c3                      retq
161d:       0f 1f 00                nopl   (%rax)
```

- objdump –d 파일이름 | grep –B3 ret 을 통해 확인 가능

- pop의 개수는 이전에 사용한 함수의 인자 수만큼 필요

- 64 bit는 인자호출 방식이 달라서 가젯이 없을 수도 있음

read(0, &bss, 8) -> system(&bss) -> exit(0)

buf	SFP	read() addr	PPR	read argv[0]	read argv[1]
		&read	&PPPR	0x0	&bss

read argv[2]	system() addr	PPR	system argv[0]	exit() addr	dummy	exit() argv[0]
0x8	&system	&PR	&bss	&exit		0x0

read(0, &bss, 8) -> system(&bss) -> exit(0)

buf	buffer	SFP	PPPR	read argv[0]	read argv[1]
			&PPPR	0x0	&bss

read argv[2]	system() addr	PR	system argv[0]	exit() addr	dummy	exit() argv[0]
0x8	&system	&PR	&bss	&exit		0x0

read(0, &bss, 8) -> system(&bss) -> exit(0)

buf	buf	buf	buf	buf	buf
		&read	&PPPR	0x0	&bss

SFP	system() addr	PR	system argv[0]	exit() addr	dummy	exit() argv[0]
0x8	&system	&PR	&bss	&exit		0x0

read(0, &bss, 8) -> system(&bss) -> exit(0)

buf	buf	buf	buf	buf	buf
		&read	&PPPR	0x0	&bss

buf	SFP	PR	system argv[0]	exit() addr	dummy	exit() argv[0]
0x8	&system	&PR	&bss	&exit		0x0

read(0, &bss, 8) -> system(&bss) -> exit(0)

buf	buf	buf	buf	buf	buf
		&read	&PPPR	0x0	&bss

buf	buf	buf	SFP	exit() addr	dummy	exit() argv[0]
0x8	&system	&PR	&bss	&exit		0x0

read(0, &bss, 8) -> system(&bss) -> exit(0)

buf	buf	buf	buf	buf	buf
		&read	&PPPR	0x0	&bss

buf	buf	buf	buf	SFP	dummy	exit() argv[0]
0x8	&system	&PR	&bss	&exit		0x0

- **GOT Overwrite**

- 함수의 got를 변경해 다른 함수가 호출되게 하는 기법

- printf의 got 값을 scanf의 주소로 바꾸면 printf 호출 시, scanf 호출

· GOT Overwrite 필요조건

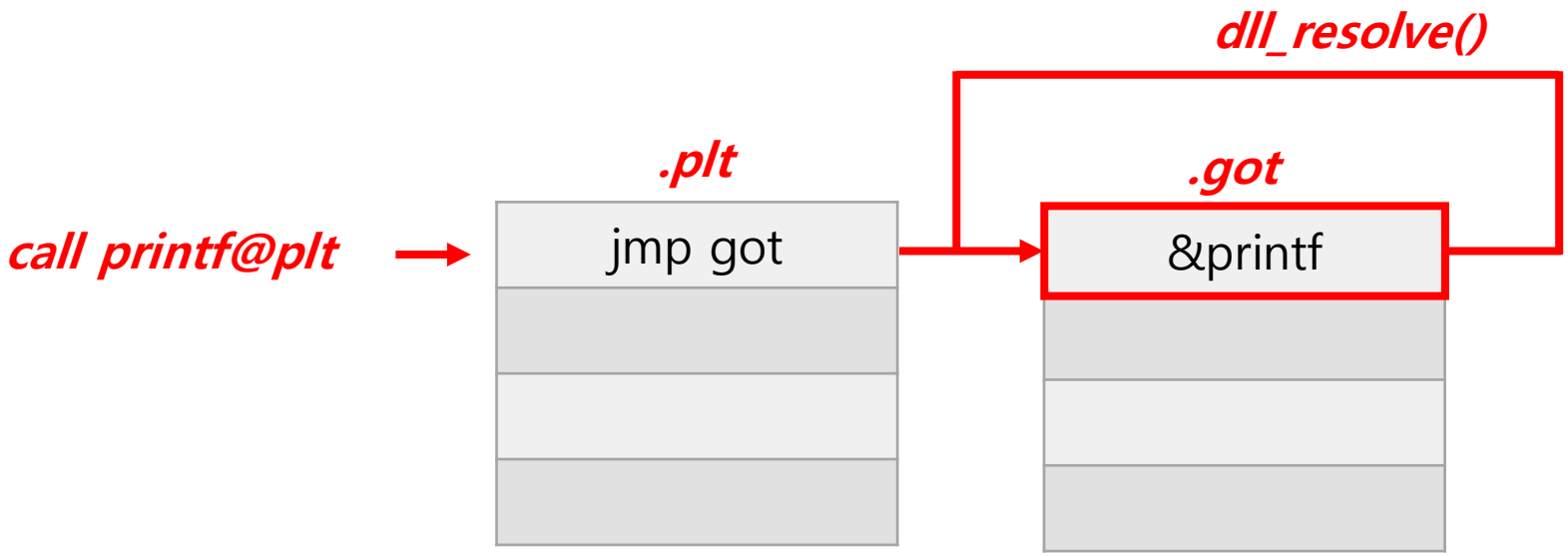
보호 기법	상태
RELRO	partial RELRO
Stack guard	No canary found
NX bit	NX enable
PIE	No PIE

1. got를 바꿀 함수 주소 획득

2. 사용할 함수 주소 획득

3. 임의 주소 쓰기 취약점 필요

· PLT와 GOT



1. got를 바꿀 함수 주소 획득

2. 사용할 함수 주소 획득

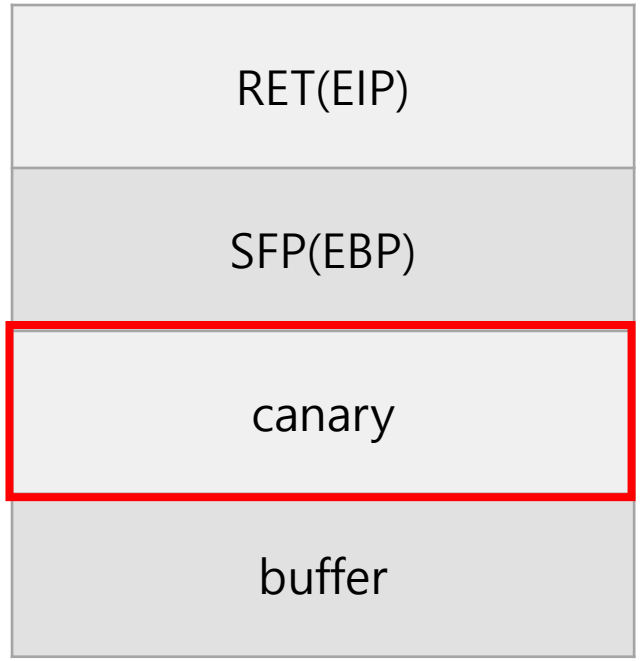
3. 임의 주소 쓰기 취약점 필요

- **Stack guard (SSP)**

- buffer와 ebp(rbp) 사이에 함수 프롤로그에서 canary라는 난수 값을 삽입

- canary 값이 변조가 되었는지 함수의 에필로그에 확인

- canary 값이 변조 되었을 시, 스택 오버 플로우가 일어난 것으로 간주하고 프로세스 종료



<프롤로그 - 생성>

```
0x0000000000001d64 <+0>:     endbr64
0x0000000000001d68 <+4>:     push   rbp
0x0000000000001d69 <+5>:     mov    rbp,rsp
0x0000000000001d6c <+8>:     sub    rsp,0x10
0x0000000000001d70 <+12>:    mov    rax,QWORD PTR fs:0x28
0x0000000000001d79 <+21>:    mov    QWORD PTR [rbp-0x8],rax
0x0000000000001d7d <+25>:    xor    eax,eax
0x0000000000001d7f <+27>:    mov    DWORD PTR [rbp-0xc],0x0
```

<에필로그 - 확인>

```
0x00000000000008e3 <+259>:   call   0x750 <__stack_chk_fail@plt>
0x00000000000008e8 <+264>:   add    rsp,0xd0
0x00000000000008ef <+271>:   pop    rbx
0x00000000000008f0 <+272>:   pop    rbp
0x00000000000008f1 <+273>:   pop    r12
0x00000000000008f3 <+275>:   ret
```