동적 링크를 활용한 특정 함수 호출

옥근호[*] · 강영진[**] · 이훈재[***]
[*]동서대학교 정보보안트랙
[**]동서대학원 유비쿼터스 IT
[***]동서대학교 컴퓨터공학부

Exploit the method according to the function call

OK Geun Ho[*] · Young-Jin Kang[**] · HoonJae Lee[***]
[*]Dept. of Information and Communication Engineering, Dongseo University
[**]Dept. of Ubiquitous IT Graduate School of Dongseo University
[***]Dept. of Computer Engineering, Dongseo University
E-mail : artist2@naver.com, rkddudwls55@gmail.com, hjlee@dongseo.ac.kr

요 약

본 논문에서는 바이너리 프로그램에서 함수가 호출될 시 바이너리 내에서 어떠한 방법으로 함수를 호출하는지 설명한다. 그리고 그 함수를 호출할시 필요한 요소들과 C언어 파일의 동적링크 컴파일 과정과 그 요소들을 이어주는 ‘링커’라는 개념을 설명하고, 정적링크와 동적링크를 차이점을 비교 분석한다. 또한 동적 링크를 활용하여 취약점을 공격하는 Return To Dynamic Linker에 대해 간략히 서술하며 테스트바이너리에 시험해본다.

ABSTRACT

In this paper, binary in the program function is to be called binary explain the function in any way to call with in the binary. And the functions required during the call to the elements and their dynamic links in the compilation process and its elements and C-language file describes the concept of ‘linker’ that connects, and static links and dynamic link Compare analysis differences. Also Do an experiment on Return To Dynamic Linker exploit.

키워드

PLT, GOT, Dynamic Link, Return To Dynamic Linker, Exploit

Ⅰ. 서 론

시스템해킹을 공부해본 사람들이라면 GOT, PLT가 뭘 나타내고 어떤 역할을 하는 지 알 수 있을 것이다. 하지만 GOT와 PLT를 연결하는 Linker와 그 Link를 이용한 Exploit 공격기법인 Return- To-Dynamic-Linker 이하 RTDL은 잘 모르는 사람들이 많을 것이다. 그런 사람들을 위해 GOT, PLT와 Linker에 대해 자세히 알아보기 위해 작성되었다.

Ⅱ. PLT, GOT 기본개념

PLT의 풀 네임은 (Procedure Linkage Table)로서 외부 프로시저를 연결해주는 테이블이다. PLT를 통하여 다른 라이브러리에 있는 프로시저를 호출하여 사용할 수 있다.

GOT의 풀 네임은 (Global Offset Table)로서 PLT가 참조하는 테이블이다. 프로시저들의 주소가 담겨있으며 실제 라이브러리가 실행될 주소를 가진 테이블이다.

여기서 함수(라이브러리)를 호출하게 되면(PLT를 호출) 하고 처음 호출시 GOT로 점프를 하여 ‘어떠한 과정’을 통해 함수의 주소를 찾아낸다. 두 번째 호출부터는 첫 번째 호출 때 알아낸 주소로 바로 점프한다.

Ⅲ. static·dynamic link 컴파일 과정

```
[gate@localhost gate]$ cat test.c
#include<stdio.h>

int main(){

        printf("hello,world\n");

return 0;
}
[gate@localhost gate]$
```

그림 1. Test code

static link는 정적링크이고 dynamic link는 동적링크라 불린다. 정적링크와 동적링크의 차이는 뭘까? 다음 [그림 1]의 간단한 소스를 컴파일 한다고 생각 해 보자.

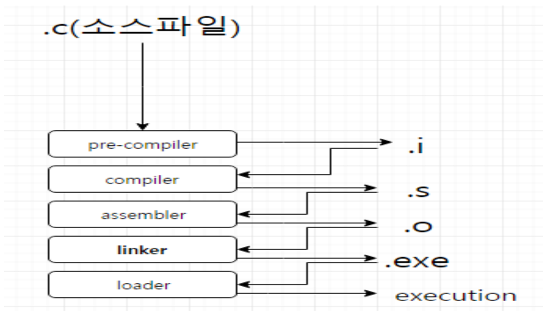

그림 2. 컴파일 과정

[그림 2]는 .c파일의 컴파일 과정이다. c파일이 여러 컴파일러와 어셈블러, 그리고 링커를 거쳐서 실행파일로 만들어지는 과정이 보인다.

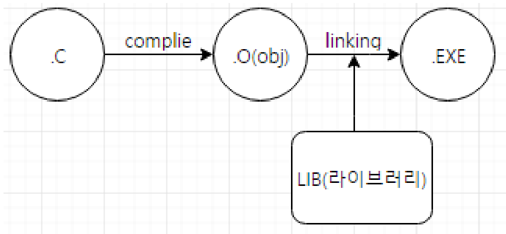

그림 3. 라이브러리가 링크되는 모습

여기서 .o(오브젝트)파일이 생성된 후 실행파일로 변하기 전에 linker(링크)를 거치게 되는데 이때, linker의 역할이 [그림 3]과 같이 printf함수 같은 함수들을 포함한 라이브러리와 .o파일을 이어서 실행파일을 만드는 것이다.

이제 static link 와 dynamic link의 차이점을 알아보도록 하겠다.

static link는 앞서 말했듯이 정적 링크로써 gcc 컴파일러 옵션 중 –static 옵션을 줘서 정적 컴파일을 할 수 있다. 정적 파일로 컴파일을 하게 되면, 실행파일 안에 모든 라이브러리코드가 포함되기 때문에 라이브러리 연동 과정이 따로 필요 없고, 한번 생성한 파일에 라이브러리를 따로 관리하지 않아도 되기 때문에 편리하다는 장점이 있다. 대신에 파일크기가 커지며 동일한 라이브러리를 사용 하더라도 해당 라이브러리를 사용하는 모든 프로그램들은 라이브러리의 내용을 메모리에 매핑 시켜야 하기 때문에 상대적으로 프로그램이 메모리를 많이 잡아 먹는다.

dynamic link 방식은 일반적인 컴파일 링크 방식으로써 공유 라이브러리를 사용한다. 라이브러리 자체를 한 메모리 공간에 매핑하고 여러 프로그램에서 공유하여 사용하는 것이다. dynamic link는 실행파일 안에 라이브러리 코드를 포함하지 않기 때문에 파일크기가 정적링크로 컴파일 된 파일보다 적을뿐더러 메모리도 적게 잡아먹는다. 또한 라이브러리도 따로 업데이트 시켜줄 수 있어 효율적인 방법이지만, 실행파일이 라이브러리에 의존해야 하기 때문에 라이브러리가 없으면 실행 자체가 불가능하다는 단점이 있다

Ⅳ. Dynamic link와 PLT, GOT의 관계

static link와 dynamic link의 특징을 알아보니 PLT와 GOT가 어느 정도 더 이해가 갈 것이다. static link로 컴파일 되면 라이브러리가 프로그램 내부에 있기 때문에 따로 연결을 해줄 필요가 없지만, dynamic방식으로 컴파일하면 라이브러리가 프로그램 외부에 있기 때문에 함수주소를 불러오는 과정이 필요한 것이다. 그 과정을 PLT, GOT가 하는 것이다.

이제부턴 Dynamic link방식의 프로그램이 함수를 호출할 때 이루어지는 과정을 자세하게 알아보도록 하겠다.

Dynamic link방식 프로그램이 함수를 호출 할 때 '첫 번째 호출' 이라면, PLT를 참조하고, PLT에서는 GOT로 점프 하는 주소를 담고 있어서 점프한 후 GOT로 점프하기 전에 Linker가 _dll_runtime_resolve라는 함수를 사용(참조)하여 필요한 함수의 주소를 찾아 GOT에 그 주소를 씌어준 후 GOT에서 그 함수를 호출하게 된다. 두 번째 호출부터는 이미 함수가 씌어져 있으니 Link의 과정을 생략하고 바로 GOT로 점프하게 되는 것이다.

```
   0x08048e54 <+16>:    call   0x804f670 <puts>
   0x08048e59 <+21>:    mov    eax,0x0
   0x08048e5e <+26>:    leave
   0x08048e5f <+27>:    ret
End of assembler dump.
gdb-peda$ p puts
$1 = {<text variable, no debug info>} 0x804f670 <puts>
gdb-peda$ r
Starting program: /home/art1sts/바탕화면/test1
test
[Inferior 1 (process 3146) exited normally]
Warning: not running or target is remote
gdb-peda$ p puts
$2 = {<text variable, no debug info>} 0x804f670 <puts>
gdb-peda$
```

그림 4. 정적링크로 컴파일 된 파일

```
   0x0804842d <+16>:    call   0x80482f0 <puts@plt>
   0x08048432 <+21>:    mov    eax,0x0
   0x08048437 <+26>:    leave
   0x08048438 <+27>:    ret
End of assembler dump.
gdb-peda$ p puts
$1 = {<text variable, no debug info>} 0x80482f0 <puts@plt>
gdb-peda$ r
Starting program: /home/art1sts/바탕화면/test
test
[Inferior 1 (process 3167) exited normally]
Warning: not running or target is remote
gdb-peda$ p puts
$2 = {<text variable, no debug info>} 0xb7e7b650 <_IO_puts>
gdb-peda$
```

그림 5. 동적링크로 컴파일 된 파일

[그림 4], [그림 5]를 보면 정적링크로 컴파일 된 파일은 함수호출시 주소가 바뀌지 않지만, 동적링크로 컴파일 된 파일은 함수호출 및 실행 시 주소가 바뀌어져 있다.

Ⅴ.Return-TO-Dynamic-Linker

```
1  #include <stdio.h>
2  #include <memory.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  int main() {
7    char *rbuf = (char*) malloc(1024);
8    int len = 0;
9    char buf[256];
10   len = read(0, buf, 1024);
11   memcpy(rbuf, buf, len);
12   return 0;
13 }
```

그림 6. RTDL 테스트 코드파일

[그림 6]은 테스트 코드를 작성을 나타낸다.

Linux art1sts-virtual-machine 4.2.0-27-generic #32~14.04.1-Ubuntu SMP UTC 2016 i686 i686 i686 GNU/Linux 환경에서 진행 되었으며 컴파일 옵션은 gcc exp2.c -o exp2 -m32 -fno-stack-protector -fno-pie를 줌으로써 pie, ssp를 해제시켰다.

```
   0x080484a7 <+42>:   mov    DWORD PTR [es
   0x080484af <+50>:   lea    eax,[esp+0x18
   0x080484b3 <+54>:   mov    DWORD PTR [es
   0x080484b7 <+58>:   mov    DWORD PTR [es
   0x080484be <+65>:   call   0x8048330 <re
   0x080484c3 <+70>:   mov    DWORD PTR [es
   0x080484ca <+77>:   mov    eax,DWORD PTR
   0x080484d1 <+84>:   mov    DWORD PTR [es
   0x080484d5 <+88>:   lea    eax,[esp+0x18
   0x080484d9 <+92>:   mov    DWORD PTR [es
   0x080484dd <+96>:   mov    eax,DWORD PTR
   0x080484e4 <+103>:  mov    DWORD PTR [es
   0x080484e7 <+106>:  call   0x8048340 <me
   0x080484ec <+111>:  mov    eax,0x0
```

그림 7. read함수의 plt

[그림 7]을 보면, read함수에서 call하는 0x8048330로 가면 read@plt가 보인다. 그리고 read@plt가 어떠한 주소로 점프하고 있고 plt+6에서 어떤 값(0x0)을 push해주고 있다. 그리고 plt+11에서는 0x8048320 으로 점프하고 있다. 여기서 0x8048320은 plt 자체의 함수 주소이다. 먼저 0x804a00c로 가서 알아보겠다. [그림 8] 은 read@got의 일부분이다.

```
.got.plt:0804A00C dword_804A00C   dd 804A03Ch
```

그림 8. read@got 일부분

0x804a00c에서 0x804a03C부분은 push 0x0을 하는 부분이다. [그림 9]에서 0x8048320을 보겠다.

```
gdb-peda$ x/2i 0x8048320
   0x8048320:   push   DWORD PTR ds:0x804a004
   0x8048326:   jmp    DWORD PTR ds:0x804a008
```

그림 9. 0x8048320주소

실제 got값을 push하고 got+4의 주소로 jump한다. got+4의 주소에는 _dll_runtime_resolve함수가 들어있다.

```
1   godists@ubuntu:~/Desktop$ readelf -d exp2
2
3   Dynamic section at offset 0xf14 contains 24 entries:
4     Tag        Type                         Name/Value
5    0x00000001 (NEEDED)                     Shared library: [libc.so.6]
6    0x0000000c (INIT)                       0x80482f4
7    0x0000000d (FINI)                       0x8048574
8    0x00000019 (INIT_ARRAY)                 0x8049f08
9    0x0000001b (INIT_ARRAYSZ)               4 (bytes)
10   0x0000001a (FINI_ARRAY)                 0x8049f0c
11   0x0000001c (FINI_ARRAYSZ)               4 (bytes)
12   0x6ffffef5 (GNU_HASH)                   0x80481ac
13   0x00000005 (STRTAB)                     0x804823c
14   0x00000006 (SYMTAB)                     0x80481cc
15   0x0000000a (STRSZ)                      88 (bytes)
16   0x0000000b (SYMENT)                     16 (bytes)
17   0x00000015 (DEBUG)                      0x0
18   0x00000003 (PLTGOT)                     0x804a000
19   0x00000002 (PLTRELSZ)                   40 (bytes)
20   0x00000014 (PLTREL)                     REL
21   0x00000017 (JMPREL)                     0x80482cc
22   0x00000011 (REL)                        0x80482c4
23   0x00000012 (RELSZ)                      8 (bytes)
24   0x00000013 (RELENT)                     8 (bytes)
25   0x6ffffffe (VERNEED)                    0x80482a4
26   0x6fffffff (VERNEEDNUM)                 1
27   0x6ffffff0 (VERSYM)                     0x8048294
28   0x00000000 (NULL)                       0x0
```

그림 10. 파일의 dynamic section 확인

이제는 push 0x0에 대해 알아 볼 건데 그전에 ELF파일의 dynamic section을 확인 해 보면 elf dynamic section은 기본적으로 [tag][type][value]로 되어있다. [그림 10]처럼 이름이 붙어지며 실행 시 배열 형태로 저장한다. 예를 들자면 배열 [17] = 0x80482cc(JMPREL) 이런 식으로 나타내어진다. push 0x0은 reloc_offset이란 값이다. reloc_offset은 _dll_runtime_resolve함수의 첫 번째 인자(0)라고 정의 해둔다. _dll_runtime_resolve 함수는 스택에 있는 두 인자 reloc_offset, 1을 각각 push한 순으로 edx, eax에 옮긴 후 dll_fixup 함수를 호출한다.

_dll_runtime_resolve(int reloc_offset, struct

link_map *l)라고 생각하면 된다. 그리고 JMPREL + reloc_offset주소에는 ELF32_Rel형식의 구조체가 담기는데, 여기서 JMPREL의 주소는 0x80482cc이다. 여기에 reloc_offset의 값 0x0을 더해도 그대로 0x80482cc일 것이다. 이 바이너리의 ELF32_Rel형식의 구조체 주소는 0x80482cc이다.

```
Relocation section '.rel.plt' at off
 Offset     Info    Type
0804a00c  00000107 R_386_JUMP_SLOT
0804a010  00000207 R_386_JUMP_SLOT
0804a014  00000307 R_386_JUMP_SLOT
```

그림 11. Elf32_Rel 값

[그림 11]을 보면 0x804a00c(read@got의 일부분)와 숫자(107, 207, …)가 보인다. 이 숫자들은 07 부분은 재배치타입을 뜻하고, 1부분은 SYMTAB의 로딩 정보번호를 뜻한다. 가령 0x00000107에서 000001은 SYMTAB 상의 번호, 07은 형식인 것이다. [그림 10]을 보면 SYMTAB의 주소는 0x80481cc이다. SYMTAB이 로딩 되는 방식은 ((ELF32_SYM*)SYMTAB)[Number]이 된다.

그러면 0x80481cc(SYMTAB)+ 1*16(ELF32_SYM의 크기)가 실제 위치가 된다.

```
gdb-peda$ x/16x 0x80481cc+16
0x80481dc:  0x1a  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x80481e4:  0x00  0x00  0x00  0x00  0x12  0x00  0x00  0x00
```

그림 12. 0x80481cc+16의 위치

[그림 12]는 ELF32_SYM 구조체 부분이다. 0x80481dc Elf32_sym<0x1a, 0, 0, 12h, 0, 0>이런 식으로 되어 있다. 0x1a부분은 STRTAB에서 부터의 함수이름 위치가 되고, 두 번째 세 번째의 0부분은 아무 값이나 집어넣어도 되는 더미 값이다. 네 번째 0x12도 중요한의미를 가지고 있지 않다. 중요한건 다섯 번째 0부분인데, 이 값에 &3 연산을 하여 0이 아닐 시 이미 로딩이 되어있다고 판단하고 호출을 해버린다. 이제 STRTAB+0x1A로 가보면 [그림 13]과 같이 read가 보이는 게 보인다. 이때까지 알아온 정보로 원하는 함수를 호출하는 exploit을 작성 해 보자.

```
gdb-peda$ x/s 0x804823c+0x1A
0x8048256:      "read"
```

그림 13. 0x804823C+0x1A 확인

기본적으로 버퍼 오버 플로 취약점을 가진 프로그램이고 memcpy 대상 포인터 조작이 가능하다. exploit에서 push 0x0 인자를 다른 값으로 변형시키고 점프를 하게 만든다. 그 인자를 따라가면 Elf32_Rel구조체가 있어야하며 Elf32_Sym이 SYMTAB[number>>8]에 구조체로 들어 있어야 할 것이다. 이과같이 익스플로잇을 작성하면 취약점을 공격할 수 있다.

Ⅵ. 결론

이때까지 Dynamic-Linker를 통한 바이너리 exploit을 알아보았다. 취약점과 exploit방법을 알아 감으로써 개발자라면 취약점을 최대한 내지 않게 개발하고, 해커라면 숨겨진 취약점을 잘 캐치해서 바이너리를 해킹한 후 취약한 부분을 잘 분석 하도록 한다.

감사의 글: 이 논문은 2015년도 정부(교육과학기술부)의 재원으로 한국연구재단의 기초연구사업 지원을 받아 수행된 것임(과제번호: 2011-0023076). 또한 부산광역시에서 지원하는 BB21 과제에서 지원받았음

참고문헌

[1] 진모씨 잡식창고, “Return-to-DL (Dynamic Linker) Exploitation”, URL : gooverto.tistory.com/39

[2] Hackerz on the ship, “PLT와 GOT자세히 알기”, URL : https://bpsecblog.wordpress.com/2016/03/07/got%EC%99%80-plt-%EC%9E%90%EC%84%B8%ED%9E%88-%EC%95%8C%EA%B8%B0-1/, https://bpsecblog.wordpress.com/2016/03/09/plt%EC%99%80-got-%EC%9E%90%EC%84%B8%ED%9E%88-%EC%95%8C%EA%B8%B0-2/